提供各种IT类书籍pdf下载，如有需要，请QQ:2404062482

注：链接至淘宝，不喜者勿入！ 整理那么多资料也不容易，请多多见谅！非诚勿扰！

更多此类书籍

VISIT…

LANZAROTE Caliente.COM

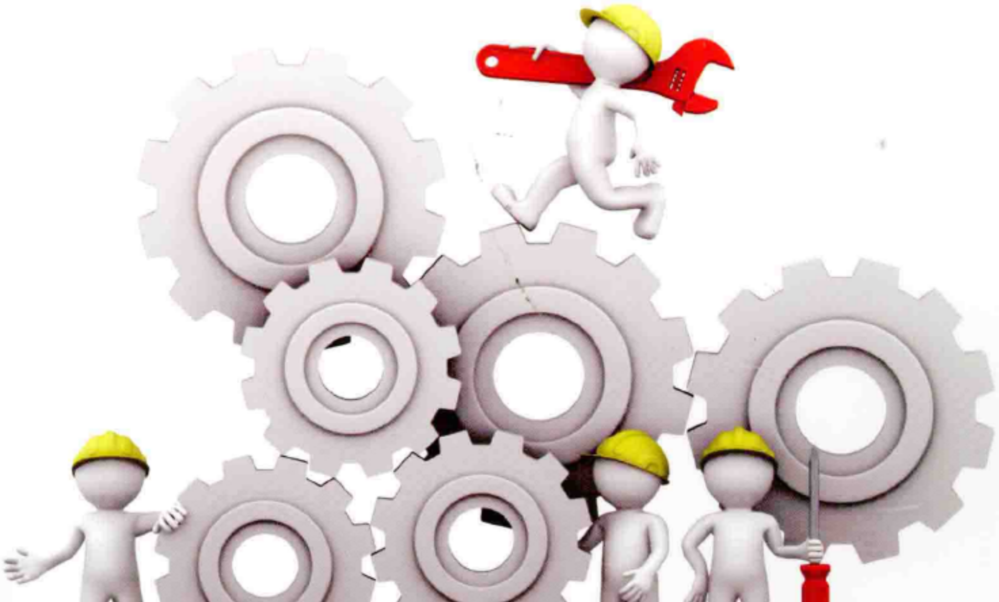
WILEY

（第3版）

Assembly Language Step-By-Step: Programming with Linux 3rd Edition

汇编语言：基于Linux环境

Jeff Duntemann 著 梁晓晖 译



清华大学出版社

汇编语言：基于 Linux 环境(第 3 版)

(美) Jeff Duntemann　著
梁晓晖　译

清华大学出版社
北 京

内 容 简 介

本书是风靡美国的经典汇编语言畅销书籍的最新版，美国计算机领域著名作者Jeff Duntemann的力作。作者以其渊博的专业知识，丰富的实战经验，结合生动详尽的实例，全面系统地介绍了Linux环境下如何使用汇编语言进行程序设计以及与之有关的背景知识和相关工具的使用。本书写作风格独特，全书采用作者最有特色的对话式风格，结合大量源于生活的暗喻，将晦涩难懂的知识点条分缕析地呈现出来，以便读者能以轻松愉快的心情学习。

本书适合刚涉足Linux环境下汇编语言的读者，也可作为相关技术人员的参考书。

Assembly Language Step-By-Step: Programming with Linux by Jeff Duntemann
EISBN: 978-0-470-49702-9
Copyright © 2005 John Wiley & Sons Ltd.
All rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except under the terms of the Copyright, Designs and Patents ACT 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd., 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher. Requests to the Publisher should be addressed to the Permissions Department, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, or emails to permreq@wiley.co.uk, or faxed to (+44)1243 770571.
All Rights Reserved. This translation published under license.

本书中文简体字版由Wiley Publishing, Inc.授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记　　图字：01-2010-1116

本书封面贴有John Wiley & Sons公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

汇编语言：基于Linux环境(第3版)/(美)达特曼(Duntemann, J.)著；梁晓晖译. —北京：清华大学出版社，2014

书名原文：Assembly Language Step-by-Step: Programming with Linux
ISBN 978-7-302-34592-3

Ⅰ. ①汇…　Ⅱ. ①达…　②梁…　Ⅲ. ①汇编语言—程序设计　②Linux操作系统　Ⅳ. ①TP313 ②TP316.89

中国版本图书馆CIP数据核字(2013)第284546号

责任编辑：文开琪
装帧设计：杨玉兰
责任校对：王　晖
责任印制：何　芊

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载：http://www.tup.com.cn，010-62791865
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×230mm　　**印　　张**：36.75　　**字　　数**：675千字
版　　次：2014年2月第1版　　**印　　次**：2014年2月第1次印刷
印　　数：1～3000
定　　价：99.00元

产品编号：034967-01

序

汇编网技术专家组

信息已经充满我们这个时代的每个角落，先放下分辨有效信息还是无效信息这件事，我们马上就能看到我们获得信息的手段、数量和速度都不断迈入新的量级。在这背后提供支撑的是数以亿万计的软件和硬件，实现这些软件和硬件的技术已经形成一个庞大的领域，这个领域就是计算机科学与技术领域。成千上万的喜欢、热爱这个领域的工作者奉献着自己的绵薄之力，对这个领域进行不断的创新和改变。我们看到，这份绵薄之力，已经形成移动互联新时代里声势浩荡的华彩乐章。

有一门语言，是任何一名想在计算机科学与技术领域发展的人必须掌握的，这门语言就是汇编语言；有一门技术，是任何一名想在计算机科学与技术领域发展的人所必须熟悉的，这门技术就是应用汇编语言的技术。我们汇编网致力于帮助学习者更好地学习与应用汇编语言，并将其作为一件重要的事去实现。

这本书是美国计算机科学与技术领域中的一本经典技术著作，很高兴清华大学出版社能够出版这本书的中文版。这一举措，为我国相关领域的读者提供了更丰富的学习方式和技术内容。同时，我们汇编网也决定以我们所持有的教学理念为前提，通过汇编网(www.asmedu.net)这个公益空间与平台，向这本书的读者提供相关的学习支持。让我们与出版社、汇编网、广大读者一起，共同为我国相关领域的发展贡献我们的力量。

以此为序。

译者序

梁晓晖

Linux 自问世以来，受到广大计算机爱好者的喜爱，主要原因有两个。一是它属于自由软件，用户不用支付任何费用就可以获得它和它的源代码，并且可以根据自己的需要对它进行必要的修改，无偿使用，无约束地继续传播。另一个原因是，它具有 Unix 的全部功能，任何使用 Unix 操作系统或想要学习 Unix 操作系统的人都可以从 Linux 获益。

当前，介绍 Linux 下汇编编程的书籍相对较少，而能够堪称经典且在美国畅销不衰的书籍更是屈指可数，本书即为其中最引人瞩目的一本。本书的作者 Jeff Duntemann 曾经使用汇编语言开发过大量的软件，具有丰富的实践经验，并撰写过多本关于汇编语言编码和优化的书籍。我希望这本书也能为我国计算机领域的读者的学习和工作贡献力量。

对于作者渊博的知识和极度认真的态度，以及为了以最容易让人们接受的方式来阐明一个知识点所花费的良苦用心，我深表敬意。

而我感触颇深的是全书独特的写作风格。Jeff Duntemann 完全采用生活化的语言，以轻松的对话方式对 Linux 汇编编程的相关知识以及学习方法进行了生动细致的讲解。全书中有大量的基于生活的暗喻，为读者更好地理解相关知识提供了多个视角以及更低的门槛。以生活化的语言来撰写一度令人觉得枯燥乏味的底层计算机语言类技术书籍，本书的确另辟蹊径。

作为畅销书的新版本，在保持上一版精华内容的同时，本书进行了大量修订。全书从相关软件基础(如计算机数制系统)，到软件编码思想(如利用过程和宏解决程序的复杂性)，以及后续扩展(如 C 库函数调用)等方面，对 Linux 下汇编语言编程等进行了全方位的介绍，为学习者尤其是初学者奠定了坚实、广泛的基础。

在翻译过程中，感谢我的家人对我的诸多关怀和帮助，在此尤其要感谢我的婆婆，要不是她老人家为我腾出宝贵的时间，本书的翻译工作恐难如期完成！

还要感谢清华大学出版社的各位编辑，是她们为本书的完成提供了大量宝贵的指导和建议，保证了本书的顺利出版！

由于水平有限，翻译中有不妥或错误之处在所难免，敬请广大读者批评指正。

前言

“你为什么要做那件事情？”

那是 1985 年，我和其他几个焦躁不安的媒体大腕一起，乘坐一辆出租车去纽约参加记者招待会。那时我刚刚开始媒体生涯(作为 *PC Tech* 杂志的技术编辑)，我的第一本书还有几个月就要出版。我碰巧坐在一个已经成名的编程栏目的作者/大师旁边。我们谈论了很多这样那样的事情，他给我留下了深刻的印象。我不会说出他的名字，因为他已经在这个领域做了很多事情，如果他不把戒烟当成第一要务的话，肯定会作出更多的贡献。

但是，碰巧我是一个 Turbo Pascal 的狂热爱好者，我真正想做的事情是要学习如何编写能够使用全新的 Microsoft Windows 用户界面的 Turbo Pascal 程序。在谈到下面这个臭名昭著的问题之前，他皱了皱鼻子，做了个鬼脸苦笑道：

“你为什么想做那件事情呢？”

我以前从来没有听到过这个问题(尽管后来不止一次地听到它)，所以有点猝不及防。为什么？因为，嗯，因为……我想知道它的工作原理。

“嗨！那是 C 语言要做的事情。”

更深一层的讨论使我彻底对 Pascal 迷失了方向。但是通过一番寻根究底之后，我了解到，不能用 Turbo Pascal 编写 Windows 应用程序。这样做是不可能的。或者这个编程栏目作者/大师根本不知道怎么做，或者二者兼而有之。我不知道真相属于哪一种，但是我的确知道了那个臭名昭著的问题的含义。

各位看官，注意：当某些人问你“为什么你想做那件事情？”，它的真实含义是：“你在问我如何做一件或者通过使用我喜欢的工具无法实现，或者彻底位于我的经验之外的事情，但是我又不想承认这一点，以免丢脸。所以……如何和那些黑鹰队较量？”

多年以来，我一遍又一遍地听到这句话：

问：如何创建一个不用扫描就能读出它的长度的 C 字符串？

答：你为什么想做那件事情？

问：如何用汇编语言编写一个能够在 Turbo Pascal 中调用的子程序？

答：你为什么想做那件事情？

问：如何用汇编语言编写 Windows 应用程序？

答：你为什么想做那件事情？

你明白了吧。对于这个臭名昭著的问题的答案通常是一样的，如果哪个狡猾的人这样问你，你应该立即以最快的速度反驳道：*因为我想知道它的工作原理。*

这是一个很不充分的回答。我每次回答这个问题都用这个答案，但是有一次例外，那是很多年以前，我宣布要写一本教那些初次接触编程的人们如何用汇编语言来编写程序的书。

问：天哪，你为什么想做那件事情？

答：因为要想具备理解所有其他编程体系工作原理的本事，这是现有的最佳方法。

对于程序员而言，有一件事情高于一切，那就是理解事情的工作原理。而且，学习如何成为一名程序员的过程几乎就是一个掌握事情工作原理的过程。这可以在各个级别上完成，取决于你所使用的工具。如果你正在用 Visual Basic 编程，你必须理解某些事情的工作原理，但是这些事情总体来说局限于 Visual Basic 本身。Visual Basic 在程序员和计算机之间放置了一个“层”，这个层隐藏了很多机制(同样的情况适用于 Delphi、Java、Python 以及很多其他的高级语言编程环境)。如果你在使用 C 编译器，那么你距离计算机更近了一些，你将看到更多的机制——而且必须看到，只有了解了这些机制的工作原理才能够正确地使用它。然而，即便如此，还有一些内容是隐藏的，哪怕是对于那些老练的 C 程序员而言。

反过来，如果你在使用汇编语言，那么你与计算机的距离就最大限度地缩短了。汇编语言什么都没有隐藏，而且也没扣留任何权利。当然，从另一方面来讲，由于在你和机器之间没有任何神奇的“层”来替你“照顾”某些事情，所以再无法为自己的无知找到任何托词。如果不知道事情的工作原理，那么你死定了——除非你拥有的知识够多，能自己把它弄明白。

我创作本书的目的不完全是为了教你汇编语言本身，这是关键所在。如果说本书有一个根本的指导思想，那就是：引入一种受过训练的对于机器的好奇心，以及一些基本的上下文，根据这些上下文，你能够从最底层开始探索计算机，并具备竭尽所能掌握它的信心。这是一件困难的事情，但是只要专注些，耐心些，没有什么你掌握不了的，而且只要有时间(我得提醒一下)，可能相当长。

事实上，这里我真正想教给你的是如何学习。

你需要准备些什么

要想按照我教的方式编写程序，你需要一台运行 Linux 的基于 Intel x86 的计算机。本书的正文和示例都假设至少运行在 386 上，但是因为 Linux 本身至少需要 386 环境，所以你已经满足硬件要求了。

你需要熟悉使用 Linux。我不能在本书中教你如何安装和运行 Linux，不过在某些不太直观易懂的情况下，我会提供一些暗示。如果你对 Linux 环境还没有熟悉，我建议你找本教程，借助它来进行工作。这样的书很多，但是我最喜欢的是 William von Hagen 撰写的 *Ubuntu 8.10 Linux Bible*(*Linux for Dummies* 虽然不错，但是内容不够全面)。

到底使用哪个 Linux 版本并不是很重要，只要至少基于 2.4 版的内核即可，当然，2.6 版更好一些。我用来编写本书示例程序的版本是 Ubuntu 8.10。哪一种图形用户界面并不重要，因为所有的程序都被编写为运行在纯文本 Linux 控制台下。汇编程序本身，即 NASM，也是纯文本。

我在讲解编程逻辑的时候以 Kate 编辑器为模型，它需要用到 GUI。实际上你可以使用任何你想用的编辑器。并不是程序本身需要 Kate 编辑器，而是如果你刚刚开始接触编程，或者经常使用高级语言的特定编辑环境的话，Kate 编辑器是一个很好的选择。

我在正文中引用的调试器是 Gdb，但是大多数情况下是通过它的内置 GUI 前端——Insight 来使用它。Insight 需要一个有效的 X Window 子系统，但是并不绑定到某一特定的 GUI 系统上，如 GNOME 或 KDE。

你不必事先知道如何安装和配置这些工具，因为我会在合适的时候，在相应的章节里涵盖所有必要工具的安装及配置。

注意，对于其他不是基于 Linux 内核的 Unix 实现而言，其功能的执行方式可能与基于 Linux 内核的不完全相同。例如在调用内核时，BSD Unix 就使用不同的规则。其他一些 Unix 版本(如 Solaris)我并不擅长。

学习计划

本书从基础开始讲起，我的意思是“零起点”。也许你已经入门，或者已略有心得也没关系。我对待起点问题非常慎重。我仍然相信，依次阅读本书各章并非浪费时间。复习很管用，嘿！你会发现，你了解的知识并不像想象中的那么多

(我经常这样)。

但是，如果你的时间很不充裕，下面这个学习进度安排表可以让你偷点儿懒。

1. 如果你已经理解了计算机编程的基本概念，可跳过第 1 章。
2. 如果你已经理解了除了十进制之外的其他数制(尤其是十六进制和二进制)背后的思想，可跳过第 2 章。
3. 如果你已经对计算机内部原理(内存、CPU 结构等)有了一些了解，可跳过第 3 章。
4. 如果你已经理解了 x86 内存寻址机制，可跳过第 4 章。
5. 不。停！勾掉它(第 4 点)。即使你已经理解了 x86 内存寻址机制，也请阅读第 4 章。

强调一下，这里的第 5 点是出于这样的原因：汇编语言编程是关于内存寻址的编程。如果不理解内存寻址机制，学习所有其他关于汇编的知识将毫无用处。所以，不要跳过第 4 章，无论你了解了或者认为了解了其他知识与否。从这一章开始，一直通读到本书的结尾。加载每一个示例程序，汇编每一个示例程序，运行每一个示例程序。力求理解每一个程序中的每一行代码。对什么内容都不要无条件地相信。

而且，不仅限于此。当你理解了知识之后，修改那些示例程序。试着采用不同的方式完成它们。勇于尝试我没有提到的那些内容。大胆些。这样还不够，最好更疯狂一些——这些二进制位是没有感情的，可能发生的最糟糕的事情无非就是 Linux 抛出一个分段错误，它将会破坏你的程序(也许会伤害你的自尊)，但是并不会破坏 Linux(“保护模式”不是白叫的！)。唯一可能存在的困难是：当你在尝试某些知识的时候，要了解程序不工作的原因，而且要像你了解所有它做的其他事情一样的清晰。然后将其记录下来。

最后是我追求的目标：教给你一种方法，通过这个方法，你可以理解计算机的每一个组成部分正在做的事情，以及它们是如何协同工作的。这并不意味着我本人会讲解计算机的每一个组成部分——即使我活得再长，也做不到。计算过程已经不再像以前那样简单，但是如果具备了耐心研究和实验的品质，你很有可能自己把它完成。最后，学习的唯一方法是：一切要靠自己。不能否认，你能够找到一些指导性内容——来自朋友的，来自互联网的，来自像本书一样的书籍的，但那仅仅是指导，车轴上的润滑剂而已。你必须搞清楚谁才是真正的主人，是你还是机器，然后使其成为主人。只有汇编语言程序员才真正有资格声称是主人，

这是一个值得思考的事实。

留意大写习惯

汇编语言是所有编程语言中最为独特的一种，它对于大小写的区分没有统一的标准。在 C 语言中，所有的标识符都是区分大小写的，但是，在汇编语言中，存在根本不区分大小写的汇编编译器。我在本书中用到的汇编编译器 NASM，只对程序员定义的标识符区分大小写。但是，对于指令助记符以及寄存器名却不区分大小写。

在有关汇编语言的书籍中常有一些书写习惯，其中之一就是将 CPU 指令助记符和寄存器名在正文中大写，而在散布于正文之中的源代码文件和代码片段中小写。这里，我也遵循这个习惯。在讨论部分的正文里，我会说 MOV、EAX 寄存器、EFLAGS 等。在示例代码中，我会用 mov、eax 及 eflags 表示它们。

这样做出于两个原因。

- 在正文中，助记符和寄存器需要醒目。因为在众多的一般文字之中，非常容易失去它们的线索。
- 为了阅读和学习本书之外的其他已有文档和源代码，你需要能够轻松地阅读汇编语言，无论它是大写的，小写的，还是大小写混合的。适应同一内容的不同表达方式，这一点非常重要。

这可能会激怒某些 Unix 界的人们，因为他们盲目崇拜小写字符。我事先为激怒他们这件事情而道歉，但是我仍然坚定地认为，那样做就是一种盲目崇拜，而且相当孩子气。

为什么我又到这了

无论你选择从什么地方开始学习本书，现在都是时间开始行动了。你只需记住，无论你面对的是什么，管它是什么黄鼠狼、机器，还是你自己的经验不足，脑海中首先要记住一点：你学习的目的是为了搞清楚它的工作原理。

让我们出发吧。

致谢

首先，我要感谢 Wiley 的 Carol Long 和 Brain Herrmann，是他们允许这本书再版，并确保了再版的实现，而且比上次出版耗时更短。

对于所有三个版本，我都欠 Michael Abrash 太多人情债，是他始终如一地在诸多方面为我提供非常明智的建议，尤其是那些 Intel 微体系结构之间的晦涩难解的区别。

尽管 Randy Hyde、Frank Kotler、Beth 和 alt.lang.asm 上的其余所有人没有意识到，但是他们其实已经通过各种方式为我提供了帮助，正是由于他们听取和回答来自汇编语言初学者的要求，才更有助于我决定哪些内容必须涵盖在像本书一样的书籍中，哪些不应该包含在里面。

最后，一如既往，向 Carol 致敬！是她 40 年如一日给予我绝对的支持和神圣的友谊，使我能够从事诸如本书之类的项目，并且将其进行到底。

作者简介

Jeff Duntemann，技术作家、编辑和讲师，同时也是一个出版业分析师。在他涉足技术领域的 30 年中，他曾经担任过施乐公司的程序员和系统分析员，Ziff-Davis 出版公司的技术期刊编辑，Coriolis Group Books 及后来的 Paraglyph 杂志社的编辑部主任。他目前是一名技术出版顾问，同时拥有 Copperwood 出版社(lulu.com 的按需印刷出版商)。Jeff 与妻子 Carol 住在科罗拉多州斯普林斯市。

目录

第 1 章 又一个令人愉快的星期六 1

1.1 一切尽在计划之中 1

1.1.1 步骤和测试 2

1.1.2 不止两种方式 3

1.1.3 计算机像我们一样思考 4

1.2 如果这是真实情况 4

1.3 此路不通，请绕行 5

1.3.1 Big Bux 游戏 6

1.3.2 玩 Big Bux 游戏 8

1.4 像棋盘游戏一样的汇编语言编程 9

1.4.1 代码和数据 10

1.4.2 地址 11

1.4.3 隐喻，将军 12

第 2 章 外星基数 14

2.1 新数学怪物归来 14

2.2 在火星上计数 15

2.2.1 火星数字剖析 17

2.2.2 数字基数的本质 19

2.3 八进制：绿色精怪怎样偷走 8 和 9 的 19

2.4 十六进制：解决数字的短缺 23

2.5 从十六进制到十进制，从十进制到十六进制 27

2.5.1 从十六进制到十进制 27

2.5.2 从十进制到十六进制 28

2.5.3 练习！练习！再练习 30

2.6 十六进制运算 31

2.6.1 列和进位 34

2.6.2 减法和借位 34

2.6.3 跨多列借位 36

2.6.4 意义何在 37

2.7 二进制 37

2.7.1 二进制值 39

2.7.2 为什么使用二进制 41

2.8 二进制简写方式：十六进制 42

第 3 章 摘下面具 44

3.1 RAXie，我们怎么不认识你 44

3.2 开关、晶体管和存储器 46

3.2.1 如果走陆路，就是 1 47

3.2.2 晶体管开关 47

3.2.3 难以置信的位缩水 49

3.2.4 随机访问 51

3.2.5 存储器访问时间 52

3.2.6 字节，字，双字，四字 53

3.2.7 精致的芯片排成一行 54

3.2.8 车间工长和流水线 57

3.2.9 对话内存 57

3.2.10 驾驭数据总线 58

3.2.11 车间工长的口袋..........59

3.2.12 流水线........................60

3.3 遵循计划行事的盒子..................60

3.3.1 取指和执行...................61

3.3.2 车间工长的内脏...........62

3.3.3 改变航向.......................64

3.4 是什么 vs.怎么做：体系结构和微体系结构.................................65

3.4.1 体系结构的演变...........66

3.4.2 地下室里的秘密机制....67

3.5 工厂经理....................................68

3.5.1 操作系统：角落办公室..........................69

3.5.2 BIOS：是软件，但并不软..............................69

3.5.3 多任务魔术...................70

3.5.4 内核提升.......................71

3.5.5 内核爆炸.......................73

3.5.6 计划..............................73

第 4 章 位置，位置，位置................74

4.1 内存模式的乐趣..........................74

4.1.1 16 位将带来 64K 存储空间.......................75

4.1.2 兆字节的本质...............79

4.1.3 向后兼容和虚拟 86 模式.........................80

4.1.4 16 位眼罩.......................80

4.2 段的本质....................................81

4.2.1 一个界限，而非一个位置.....................84

4.2.2 用两个 16 位寄存器构成 20 位地址.............84

4.3 16 位和 32 位寄存器....................87

4.3.1 通用寄存器...................88

4.3.2 半寄存器.......................90

4.3.3 指令指针寄存器............91

4.3.4 标志寄存器...................92

4.4 三种主要的汇编编程模型...........93

4.4.1 实模式平面模型............93

4.4.2 实模式段模型...............95

4.4.3 保护模式平面模型........97

4.5 保护模式下不再允许我们做的事情..100

4.5.1 内存映射视频系统......100

4.5.2 直接访问端口硬件......101

4.5.3 直接调用 BIOS............102

4.6 展望未来：64 位“长模式”.....102

第 5 章 汇编的权利............................105

5.1 文件及其包含的内容.................106

5.1.1 二进制文件 vs. 文本文件.....................106

5.1.2 用 Bless 编辑器查看文件内容....................108

5.1.3 解释原始数据.............112

5.1.4 “字节序”...................113

5.2 文本进去，代码出来.................116

5.2.1 汇编语言.....................117

5.2.2 注释............................119

5.2.3 当心“只写”源代码.........................120

5.2.4 目标代码和连接器......120

5.2.5 重定位能力.................123

5.3 汇编语言开发过程.....................124

5.3.1 工作目录规范 125
5.3.2 编辑源代码文件 126
5.3.3 编译源代码文件 126
5.3.4 汇编错误 127
5.3.5 回到编辑器 129
5.3.6 汇编警告 129
5.3.7 连接目标代码文件 130
5.3.8 连接错误 131
5.3.9 测试.EXE 文件 131
5.3.10 错误 vs.漏洞 132
5.3.11 我们还在那里吗 133
5.3.12 调试器和调试 133
5.4 沿着汇编小路旅行 134
5.4.1 安装软件 135
5.4.2 第 1 步：在编辑器中编辑程序 137
5.4.3 第 2 步：使用 NASM 编译程序 138
5.4.4 第 3 步：使用 LD 连接器 140
5.4.5 第 4 步：测试可执行文件 141
5.4.6 第 5 步：在调试器中观察程序运行 142
5.4.7 准备好要来真格的了吗 148

第 6 章 有地儿，有工具 149

6.1 Kate 编辑器 151
6.1.1 安装 Kate 编辑器 151
6.1.2 启动 Kate 152
6.1.3 配置 154
6.1.4 Kate 会话 156
6.1.5 Kate 编辑器的文件管理 158
6.1.6 向工具栏添加菜单项 161
6.1.7 Kate 编辑器的编辑控制 162
6.1.8 编程期间使用 Kate 编辑器 166
6.2 Linux 和终端 169
6.2.1 Linux 控制台 169
6.2.2 Konsole 中的字符编码 170
6.2.3 三个标准 Unix 文件 172
6.2.4 I/O 重定向 173
6.2.5 简易文本过滤器 175
6.2.6 使用转义序列进行终端控制 177
6.2.7 为什么不用汇编语言编写 GUI 应用程序呢 178
6.3 使用 Linux Make 179
6.3.1 依赖条件 180
6.3.2 文件何时最新 182
6.3.3 依赖链 182
6.3.4 从 Kate 编辑器内部调用 Make 实用工具 184
6.3.5 使用 touch 命令强制执行生成操作 187
6.4 Insight 调试器 187
6.4.1 运行 Insight 188
6.4.2 Insight 的众多窗口 189
6.4.3 快速体验 Insight 190
6.4.4 拿起你的工具 193

第 7 章 跟踪指令 194

7.1 为自己建立一个沙盒 194

7.1.1 一个最小的 NASM 程序 195

7.1.2 指令及其操作数 197

7.1.3 源操作数和目标操作数 197

7.1.4 立即数 198

7.1.5 寄存器数据 200

7.1.6 内存数据 202

7.1.7 混淆数据和它的地址 203

7.1.8 内存数据的尺寸 204

7.1.9 糟糕的过去 204

7.2 CPU 的标志位 205

7.2.1 标志规范 208

7.2.2 使用 INC 指令和 DEC 指令加 1 和减 1 208

7.2.3 从 Insight 中观察标志 209

7.2.4 标志如何改变程序的执行 211

7.3 有符号值和无符号值 214

7.3.1 补码和 NEG 214

7.3.2 符号扩展和 MOVSX ... 217

7.4 隐式操作数和 Mul 219

7.4.1 MUL 和进位标志 221

7.4.2 使用 DIV 实现无符号除法 221

7.4.3 x86 中的“慢动作”指令 223

7.5 阅读和使用汇编语言参考资料 223

7.5.1 对于复杂记忆的唤醒文件 224

7.5.2 初学者汇编语言参考指南 224

7.5.3 标志 225

7.6 NEG：求补(求补码；即，与-1 相乘) 225

7.6.1 合法形式 227

7.6.2 操作数符号 227

7.6.3 示例 228

7.6.4 注解 228

7.6.5 这里没有包含的内容 229

第 8 章 我们的崇高目标 230

8.1 汇编语言程序的基本框架 230

8.1.1 最开始处的注释块 232

8.1.2 .data 段 233

8.1.3 .bss 段 233

8.1.4 .text 段 234

8.1.5 标号 234

8.1.6 已初始化变量 235

8.1.7 字符串变量 235

8.1.8 通过 EQU 和$推导字符串的长度 237

8.2 通过堆栈实现后进先出 239

8.2.1 每小时 500 个盘子 239

8.2.2 堆栈的内容上下颠倒 241

8.2.3 Push-y 指令 242

8.2.4 POP 指令 244

8.2.5 临时存储 246

8.3 通过 INT80 使用 Linux 内核服务 247
8.3.1 不中断任何事情的中断 247
8.3.2 再次返回..................... 252
8.3.3 通过使用 INT 80h 退出一个程序 253
8.3.4 软件中断 VS 硬件中断 254
8.3.5 INT 80h 和可移植性盲目崇拜..................... 255
8.4 设计一个有价值的程序............. 256
8.4.1 问题定义..................... 257
8.4.2 从伪代码开始 258
8.4.3 连续改进..................... 259
8.4.4 不可避免的“哎呀！”时刻 263
8.4.5 扫描缓冲区 264
8.4.6 缓冲溢出(“Off By One”)错误 266
8.4.7 进一步学习 271

第 9 章 位、标志、分支和表 272

9.1 位就是二进制位(字节也是二进制位)................................. 272
9.1.1 位编号 273
9.1.2 逻辑操作...................... 273
9.1.3 与指令 274
9.1.4 位屏蔽 275
9.1.5 或指令 276
9.1.6 异或指令...................... 276
9.1.7 非指令 278
9.1.8 段寄存器对逻辑操作没有反应 278
9.2 移位操作 279
9.2.1 根据什么进行移位操作.............................279
9.2.2 移位指令的工作原理.............................279
9.2.3 将位移入进位标志 280
9.2.4 循环移位指令..............280
9.2.5 将已知值存入进位标志 CF 282
9.3 位操作 282
9.3.1 将一个字节分解为两个“半字节” 285
9.3.2 将高半字节移入低半字节 286
9.3.3 使用查找表 286
9.3.4 通过移位和相加来实现相乘 288
9.4 标志、测试和分支.....................291
9.4.1 无条件转移 292
9.4.2 条件转移指令..............292
9.4.3 条件“缺席”时进行跳转 293
9.4.4 标志 294
9.4.5 通过 CMP 进行比较操作 295
9.4.6 转移指令的错综复杂之处 296
9.4.7 “大于”与“以上” 296
9.4.8 使用 TEST 指令查找位 1 298

9.4.9 使用 BT 指令查找位 0 300

9.5 保护模式下内存寻址详解 301

9.5.1 有效地址计算 303

9.5.2 位移量 303

9.5.3 基址+位移量寻址方式 304

9.5.4 基址+索引寻址方式 304

9.5.5 索引×缩放比例+位移量寻址方式 305

9.5.6 其他寻址方式 307

9.5.7 LEA：最机密的数学机器 309

9.5.8 16 位寄存器的负担 311

9.6 字符表转换 312

9.6.1 转换表 312

9.6.2 用 MOV 或者 XLAT 进行转换 315

9.7 用表来代替计算 319

第 10 章 分治 321

10.1 盒子里面的盒子 322

10.2 调用和返回 331

10.2.1 调用中的调用 334

10.2.2 意外递归的危险 335

10.2.3 一个需要提防的标志规范 Bug 336

10.2.4 过程及其所需的数据 337

10.2.5 保存主调程序的寄存器 338

10.2.6 局部数据 341

10.2.7 更多的表格技巧 342

10.2.8 在过程定义中加入常量数据 344

10.3 局部标号和跳转的长度 345

10.3.1 “强行”访问局部标号 348

10.3.2 短转移、近转移和远转移 349

10.4 生成外部过程库 350

10.4.1 全局声明和外部声明 351

10.4.2 全局过程和外部过程的机制 353

10.4.3 连接库文件到程序中 361

10.4.4 太多过程和太多库的危险 362

10.5 自定义过程的艺术 362

10.5.1 可维护性和可重用性 363

10.5.2 确定哪些代码应该成为一个过程 364

10.5.3 使用注释标头 365

10.6 Linux 控制台下的简单光标控制 366

10.7 创建和使用宏 374

10.7.1 宏定义机制 375

10.7.2 定义带参数的宏 380

10.7.3 宏调用机制 382

10.7.4 宏内部的局部标号 383

10.7.5 像包含文件一样的宏库文件 384

10.7.6 宏 vs. 过程：优点和缺点 385

第 11 章 字符串奏鸣曲 387

11.1 汇编语言字符串的概念 387
11.1.1 彻底颠覆你的“字符串感觉” 388
11.1.2 源字符串和目标字符串 388
11.1.3 虚拟文本显示屏 389
11.2 REP STOSB，软件机枪 397
11.2.1 机枪扫射虚拟显示器 397
11.2.2 执行 STOSB 指令 398
11.2.3 STOSB 和方向标志(DF) 399
11.2.4 在显示缓冲区中定义行 400
11.2.5 将缓冲区发送到 Linux 控制台 401
11.3 半自动武器：不带 REP 的 STOSB 401
11.3.1 是谁递减了 ECX 402
11.3.2 LOOP 指令 402
11.3.3 在屏幕上显示一个标尺 403
11.3.4 MUL 并非 IMUL 404
11.3.5 添加 ASCII 数字 406
11.3.6 调整 AAA 408
11.3.7 Ruler 过程的教训 409
11.3.8 STOS 指令的 16 位版本和 32 位版本 409
11.4 MOVSB：快速块拷贝 409
11.4.1 DF 和重叠块移动 411
11.4.2 使用 Insight 单步调试 REP 字符串指令 413
11.5 将数据存储到不连续的字符串中 414
11.5.1 显示一个 ASCII 表 414
11.5.2 嵌套指令循环 416
11.5.3 当 ECX 变为 0 时进行跳转 416
11.5.4 关闭内层循环 417
11.5.5 关闭外层循环 418
11.5.6 Showchar 小结 419
11.6 命令行参数和堆栈检查 419
11.6.1 两块虚拟内存 420
11.6.2 Linux 堆栈剖析 422
11.6.3 为什么堆栈的地址是不可预测的 424
11.6.4 使用 Insight 设置命令行参数 424
11.6.5 通过 Insight 的内存视图查看堆栈 425
11.7 使用 SCASB 进行字符串搜索 427
11.7.1 REPNE v.s. REPE 431
11.7.2 从堆栈中弹出，还是对堆栈寻址 432
11.7.3 额外的学分 434

第 12 章 C 语言 435

12.1 什么是 GNU 436
12.1.1 “瑞士军刀”编译器 437

12.1.2 以 GNU 的方式生成代码 437
12.1.3 如何在汇编工作中使用 gcc 439
12.1.4 为什么不用 gas 440
12.2 连接到标准的 C 函数库 441
12.2.1 C 调用公约 442
12.2.2 建立一个框架 443
12.2.3 保存和恢复寄存器 443
12.2.4 建立堆栈帧 444
12.2.5 销毁堆栈帧 446
12.2.6 通过 puts()输出字符 447
12.3 使用 printf()格式化文本输出 448
12.4 使用 fgets()和 scanf()进行数据输入 452
12.5 驾驭时间 459
12.5.1 C 库的时间机制 459
12.5.2 从系统时钟中取出 time_t 值 461
12.5.3 将 time_t 值转换为一个格式化字符串 461
12.5.4 生成单独的本地时间值 462
12.5.5 通过使用 MOVSD 复制 glibc 的 tm 结构 463
12.6 理解 AT&T 指令助记符 467
12.6.1 AT&T 助记符公约 467
12.6.2 查看 gcc 创建的 gas 源文件 468
12.6.3 AT&T 内存引用语法 471
12.7 产生随机数 472
12.7.1 利用 srand()为随机生成器“播种” 473
12.7.2 产生伪随机数 474
12.7.3 有些位比其他位更具随机性 479
12.7.4 调用寄存器中的地址 481
12.8 C 如何看待命令行参数 482
12.9 简单文件输入/输出 484
12.9.1 通过 sscanf()将字符串转换为数字 485
12.9.2 创建和打开文件 486
12.9.3 使用 fgets()从文件中读取文本 488
12.9.4 使用 fprintf()写文本到文件中 490
12.9.5 关于收集过程到库中的注解 492

结论：不是结束，而是刚刚开始 501

附录 A 部分 x86 指令集 505

附录 B 字符集图 565

第1章

又一个令人愉快的星期六

了解计算机的实际用途

1.1 一切尽在计划之中

“快点快点，迈克，赶紧叫弟弟妹妹起床，都七点多了！ 尼基九点钟要去少棒联盟，迪欧妮十点钟要上芭蕾课。还有，别忘了给麦克斯吃犬心丝虫药丸！(妈妈，药丸已经用完了，不记得了吗？)你爸爸真会选时间，非挑这个周末去钓鱼。过来，我给你 10 个硬币，快去兽医那儿多买点儿药丸。上帝呀，差点忘了，为了给汉克交汽油钱，我已经分文皆无了。Kmart 附近有一个自动柜员机，如果我去那儿的话，可以把那个愚蠢的马桶垫圈拿回去，换个尺寸合适的。”

“我想我最好列个清单……”

又是一个令人“愉快”的星期六。家住在百万郊区的一位三十多岁的主妇正拿着铅笔和信笺簿坐在厨房的桌子旁，企图搞清楚一早上要必须完成哪些事情，可以砍掉哪些次要事情。她满脑子思考着某些事情的依赖关系，描绘出这样的路线：

让尼基到 Rand Park 下车，然后返回 Dempster，大概需要 10 分钟左右的时间到 Golf Mill 商业街。汽油是不是还有呢？最好先检查一下——如果没有，就在 Del’s Shell 停下，否则的话，开不到 Milwaukee 大街了。在 Golf Mill 的银行自动柜员机取些钱，然后穿过停车场到 Kmart 超市，把汉克上周末买的马桶垫圈拿去调换，那家伙根本没有检查型号。务必想着把马桶垫圈扔到面包车的后备厢里——把这个写在清单的最上面。

那时，时间已经过去半小时了，甚至更晚些。学芭蕾舞的地方刚好在 Park Ridge 的 Greenwood 的沿途。Milwaukee 没有左转弯——但是，在购物中心后面有一条小道可以绕过。我必须记住，坚决不能再像以前那样右转到 Milwaukee 了——

把它记下来。在 Park Ridge 的这段时间里，我可以核对一下汉克的新眼镜是不是已经到货——应该打电话，但是他们 9:30 才开门。对了，还需要采购点儿杂货——可以在迪欧妮跳舞的时候做这件事情。然后在回家时顺路去 Oakton 那里取回狗狗的药丸。

在大约 90 秒的时间里，她列出了以下清单：

- 把马桶垫圈扔到面包车里。
- 检查是否还有汽油——如果没有，在 Del's Shell 停车。
- 让尼基在 Rand Park 下车。
- 在 Golf Mill 的银行自动柜员机前停一下车。
- 把马桶垫圈退还给 Kmart。
- 让迪欧妮在芭蕾舞学校下车(记住要到 Greenwood 的小路)。
- 看一下汉克的眼镜是否还在 Pearle Vision 眼镜店——如果在，确认一下他们是否记得那个额外的涂层划痕。
- 从 Jewel 那里买些杂货。
- 接迪欧妮。
- 在兽医那里停一下，取回狗狗的心丝虫药丸。
- 到家后卸杂货。
- 如果已经到时间了，就去接尼基。如果时间未到，就休息几分钟，然后再去接尼基。
- 休息！

我们通常称之为“洗衣清单”(不管它是否包含洗衣)的东西其实是一个非常好的计算机程序的隐喻。在毫不知情的情况下，我们勇敢的主妇已经为她自己编写了一个计算机程序，然后(像计算机一样)着手执行它，并且在中午之前执行完毕。

计算机程序只不过是你(程序员)编写的一个步骤和测试的清单。随后，计算机执行每一步，按照顺序进行测试。当清单里的步骤执行完毕时，计算机就停止工作。

计算机程序是一个包含步骤和测试的清单，无他。

1.1.1 步骤和测试

考虑一下前面的“洗衣清单”中我称之为“测试”的内容。测试是一种二选一决策，即使是在平静如水的日常生活中，我们也可能做过数十次或者上百次的

决策，甚至不经意间就做出了某种决策。

在跳上面包车开始行程的时候，我们的主妇执行了一个测试：她看了看汽油表。汽油表会告诉她两件事情之一：她有足够的汽油，或者没有。如果有足够的汽油，那么她就右转弯，去 Rank Park。如果没有足够的汽油，那么她就左转弯，去拐角处的 Del's Shell 把油箱加满。然后，她驾驶着加满油的汽车，掉个头，接着开往 Rand Park 来继续这个程序。

抽象地讲，测试包含以下两部分内容。

- 首先，需要查看一下某些能够以两种方式之一进行工作的东西。
- 其次，根据观察到的结果，做两件事情之一。

在接近程序的末尾，我们的主妇回家了，她将杂货从面包车里卸下来，然后看了看钟表上的时间。如果还没到把尼基从少棒联盟接回来的时间，就在几乎空空荡荡的房子里，躺在沙发上歇一会儿。如果已经到了接尼基的时间，那就没有时间休息了，她得一路小跑到面包车那儿，然后调头前往 Rank Park。

(当程序运行完毕时，至于她到底休息还是没有休息，有人能猜得出来吗？)

1.1.2　不止两种方式

你也许会提出异议，因为很多或者大多数测试包含的选择并不都只是两个。哈哈，对不起，你完全错了——大错特错。而且，不管你承认还是不承认，你都错了。把这句话读两遍：*除了完全冲动的或者精神病患者的行为之外，人类的任何一个决策都可以归结为二选一。*

在进行决策的时候，必须做的事情是近距离观察脑海里所想的内容。下一次你去 Yow Chow Now 去吃中式快餐的时候，在注视菜单期间观察一下你自己。一开始，你面临的选择看起来像是要从 26 道广东主菜中选出一个来。事实却并非如此。实际上，选择是在选择某一项和不选择那一项之间进行的。你的目光停留在腰果鸡丁上。不，太没滋味了。*那就是一个测试。*你接着往下看下一道菜。冬菇炖鸡。嗯，不，上周已经吃过了。**这是另一个测试**。下一道菜：宫保鸡丁。对了，就是它了！**这是第三个测试。**

选择不是在腰果鸡丁、冬菇炖鸡和宫保鸡丁之间进行。每一道菜都有它的时间，为你挑剔的眼光做好准备。你把拇指分别放在它的上面或者下面。最后，有一道菜胜利了，但是它是在“吃或者不吃”的同一个游戏中获得胜利的。

下面我再举另外一个例子。生活中，大部分复杂的决策都源自于我们中的 99.9986755%都不是裸体主义者这一事实。举个例子：你正在内衣衣柜里，匆匆

查看短裤衣架。测试来得快速而且密集。这条？不行。这条？不行。这条？不行。这条？对了。你取下一条蓝色的短裤，假设是这样(毕竟，那是星期一，蓝色将是个比较合适的颜色)。然后你睡意未消地蹒跚到袜子抽屉边。哎哟，没有蓝色的袜子。**那就是一个测试。**所以你又睡意未消地返回到衣柜，将蓝色短裤挂回原来的衣架，又开始选择。这条？不行。这条？不行。这条？可以。这次是条棕色的短裤。你把裤子轻放到胳膊上，然后又回过头去袜子抽屉再看一眼。切！也没有棕色的袜子。所以你又回到了衣橱……

当你考虑单个决策或者也许在两个决策(例如，在现存的库存存货中，选择同样颜色的裤子和袜子)之间纠结的时候，实际上你思考的应该是一系列小的决策，本质上经常是二进制的：选或者不选，找到或者没找到。星期一早上在衣橱里发生的故事片段类似于一种名为循环的程序结构：你一直在重复地做某一系列事情，直到得到正确的结果，然后停下来(假设你不是那种穿蓝色袜子配棕色裤子的“另类”)。不管所有事情做对与否，归根结底都是一系列简单的二选一决策问题。

1.1.3 计算机像我们一样思考

我能想象到有人会这样反驳我：“当然，这是本计算机书籍，他是在尽量让我们像计算机一样思考。”不是那么回事儿。是计算机像我们一样思考。我们设计了它们，它们怎么能有其他的思考方式呢？不，我正企图做的事情是让你长时间地、努力地看一看自己是如何思考的。我们生活中有很多事情都是自动完成的，以至于我们没有真正地思考就做完了大部分应该思考的事情。

计算机程序逻辑最理想的模型与我们用来计划和管理日常事务的逻辑模型基本上一样。不管我们做什么事情，最后都可以归结为一件事情：面对两个选择，选出其中的一个。我们认为的大而复杂的决策无非是很多小决策盘根错节地纠结在一起而已。审视一个复杂的决策，并从中看出它包含的所有小决策，这项技能将为你学习如何编程提供很好的帮助。下次当你必须进行决策的时候，请观察一下自己，并算出组成那个大决策的所有小决策的总数。你会大吃一惊的！

并且，还有意外的惊喜！你将成为一名程序员。

1.2 如果这是真实情况

不要担心。刚才的体验仅仅是个隐喻。真实的情况并非如此。(后面会对真

实的情况进行讲解。)在这本书中我经常用到隐喻。隐喻是在有些熟悉(例如星期六上午的洗衣清单)，有些不熟悉(例如计算机程序)的事情之间的一个不严格的比较。其背后的思想是用熟悉的术语对那些不熟悉的事情进行阐述，目的是当我开始讲解真相的时候，你能够比较容易地理解。

现在，你需要做的最重要的事情就是时刻保持开放的思想。如果你已经对计算机和编程有所了解，不要吹毛求疵。是的，一个主妇遵循一个草草写下的洗衣清单行事和计算机执行程序，二者之间存在着重要的区别。我会在讲解之前事先提到所有这些区别。

目前，仍然是第 1 章。我在使用这些浅显的隐喻的自身术语来呈现它们。以后，它们会对我们的理解提供很大的帮助。

1.3　此路不通，请绕行

“bored 和 board 是同音词有一个原因。”一天晚上，当我们(两个超级老练的 12 岁游戏玩家)坐在我最好的朋友奥特的地下室里玩游戏时，他说道。(他那时可能不高兴了，因为他输了游戏。)那是日行千里、新陆军棋、大富翁，还是什么其他游戏？我承认，我已经记不得了。我只回想起将一些外形像大肚保龄球瓶的小塑料在一系列彩色的方格里上下跳动，那个方格告诉我做一些笨笨的事情，比如往后跳两个格，或者把 100 美元放入桶中，或者用核武器发起进攻等。

在那些独特的美国娱乐活动、棋盘游戏和汇编语言编程之间，有着很强的相似性。首先，前面说过的所有内容仍然适用：总的来说，棋盘游戏包含一系列步骤和测试。在某些游戏里，例如，*Trivial Pursuit*(一种风靡全球的棋盘问答游戏)，棋盘上的每一步都是一个测试：看一下你是否能够回答卡片上的问题。在其他的棋盘游戏中，沿着路径的每个小方格都包含一些指令类型：Lose One Turn(失败一个回合)，Go Back Two Squares(回退两个方格)，Take a Card from Community Chest(从社区福利基金中取出一个卡片)，当然，还有 Go to Jail(请进监狱)。这些事情发生在棋盘游戏里，当沿着棋盘边缘行走的时候，那些小小的大肚保龄球所采用的路径会发生变化。

很多棋盘游戏也有少许在棋盘上标识出来的存储空间，你可以用它们来存放东西：卡片、玩具钞票或者游戏道具，例如小塑料房子、塑料旅馆，或者是炸弹人、战术核导弹。在游戏过程中，你可以买、卖或者着手建设各种资产，你的存储空间的内容也随之不断发生变化。计算机程序跟它类似：也有一些存放东西的

地方(这里的“东西”指的是纯数据，而不是真正的道具)；当计算机程序执行时，存放在那些地方的数据也会发生改变。

计算机程序并不是游戏，当然——至少不是从棋盘游戏是游戏这种意义上来说。大多数时间里，任一给定程序都是自己运行的。只有一个“玩家”，而不是两个或者多个。**(这不总是对的，但是，我不想讲得太超前。记住，我们仍然处于隐喻阶段。)隐喻的用处依然很大，值得我们继续追求。**

1.3.1 Big Bux 游戏

为了继续使用这一特定的隐喻，我已经开发了一款自己的棋盘游戏。从艺术反映生活的意义上来讲，Big Bux 这个游戏反映了硅谷的生活，在那里，钱被不由自主地创造出来(一般情况下进入某些其他人的口袋)，3 个巨大的资金黑洞分别为高速行驶的汽车、加利福尼亚的不动产和狼藉的离婚事件。风水轮流转，好运、工作及不动产经常非常迅速地更换主人。

图 1-1 展示了一部分 Big Bux 游戏板。页面左侧的一列矩形展示了该游戏板中的所有路线。在游戏板的中间位置有一些用来存放游戏币和游戏卡的小孔；一堆偶尔将被读取的卡片；一些名为“Messy Divorce(麻烦的离婚)”或“Start a Business”(开始一项业务)之类的短绕路(detour)，这些绕路对沿游戏板边缘绕行的路线中的“行为方块”进行了展开描述。当接收到一些或者来自游戏板上的某个方格，或者来自游戏中取出的卡片的指示时，玩家就要走这些旁路。如果刚好到达一个“Start a Business”的方格，就走“Start a Business”那条旁路。如果跳过了那个方格，就不走那条旁路，只需绕着游戏板继续前行即可。

与很多棋盘游戏不同，这里不用通过扔骰子来决定沿着游戏板走多少步。Big Bux 要求每一回合都往前走一步，除非到达的方格指示你向前或向后或向其他地方走，例如：通过一条绕路。这样就构成了一个非常小的随机游戏。实际上，Big Bux 是一个有趣的线性体验，这意味着大部分时间你是沿着游戏板前行，直到被告知游戏结束。这时，你也许已经破产了；如果没有，你可以核算一下资产，看看玩得有多棒。

这里涉及一些数学运算。最开始的时候，你拥有一套公寓、一辆便宜的小汽车和 250000 美元的现金。你能够按照某一给定利率购买 CD，然后在每一次沿着游戏板绕行时支付利息。你可以投资股票和有价证券，其价值由经济指示器中的可变索引来决定，该指示器根据从名为“Fickle Finger of Fate”(命运多变手指)的堆中选出的卡片进行波动。你可以在二手市场卖掉汽车，买卖房产、公寓和土

地；鼓动其他玩家并与其进行交易。每当绕着游戏板做一件此类事情，你都必须重新计算一下资本净值。所有这些都包含一些加法、减法、乘法和除法，但是没有哪一种数学运算比计算复合利率更复杂了。大部分 Big Bux 游戏都不过是向前走一步和遵循每一步中的指令而已。

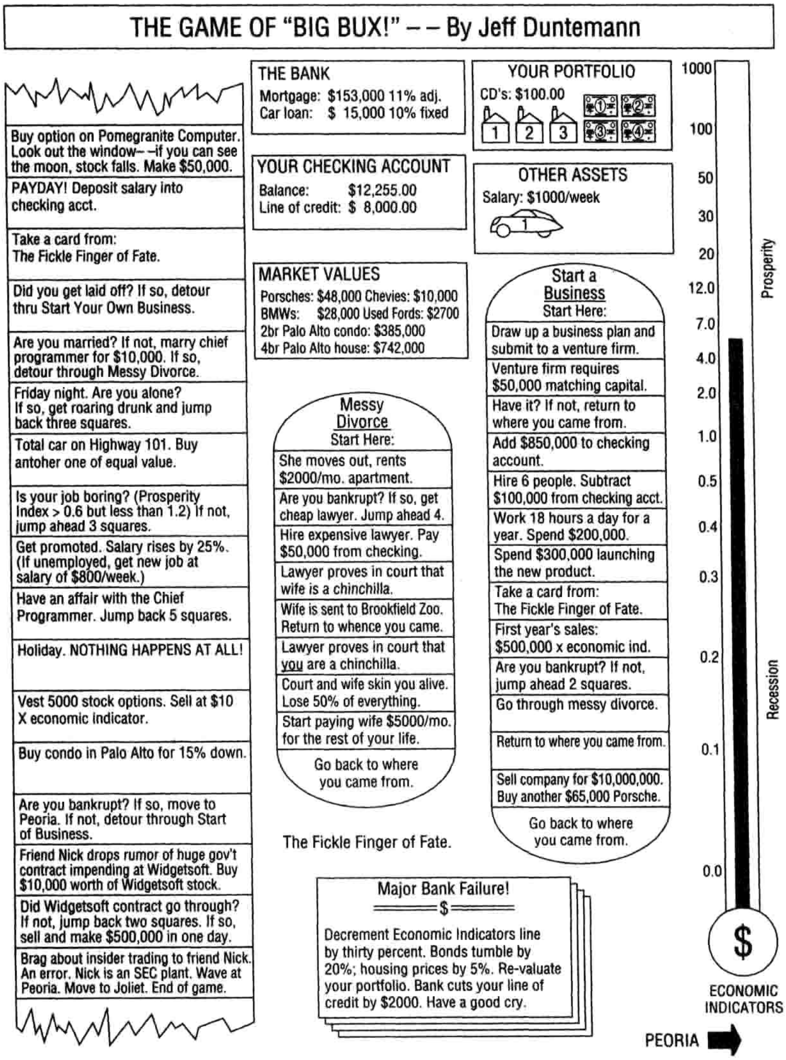


图 1-1　Big Bux 游戏板

开始觉得有点儿熟悉了吗？

1.3.2　玩 Big Bux 游戏

Big Bux 游戏板的某个角落就是传说中的“Move In”(迁入)，就像人们在加利福尼亚开始生活一样——实际上没有人是在那里出生的。这就是游戏开始的入口点。一旦迁入之后，就可以一个方格一个方格地按照方格中的指令，以自己的方式开始这个游戏了。

有些方格简单地告诉你做某些事情，例如：“以 15%的折扣在 Palo Alto 购买一栋楼。”很多方格都包含一个某种类型的测试。例如，一个方格写着这样的内容：“你的工作令人乏味吗？(繁荣指数为 0.3～4.0。)如果不，往前跳 3 个方格。”这个测试实际上是要看一下繁荣指数值是否为 0.3～4.0。在此范围之外的任意值(也就是说，远离繁荣或 4 个骑士级经济衰退)被定义为有趣的时光，将会导致往前跳 3 个方格。

每一次都往前走一步，除非到达的方格指示你做一些其他事情，比如：往前走 3 个方格，或者往回跳 5 个方格，或者绕路而行。

绕路而行的概念非常有趣。在我提供的部分游戏板上显示了两条绕路。(整个游戏板还有一些其他绕路。)“绕道而行”意味着离开(沿着游戏板边缘绕行的)主路，然后进入游戏板上其他地方的一系列方格。绕行完成之后，再回到原来的路线，而且刚好返回到离开时所在的地方。“绕道而行”包含某些特定的过程——例如，Start a Business(开始一项新的业务)，或者 Messy Divorce(离婚了)。

你可以一步一步地穿过一个绕路，直到到达最深处。那时，你只需简单地回到离开的主路，继续旅行。你也可能会发现，“绕道而行”过程中的某个方格将指示你原路返回。根据游戏的逻辑(以及你的运气和财政情况)，你可能完全通过这条绕路，也可能从该绕路的中间跳出。无论哪种情况，你都会返回最初进入该绕道的那个地点。

还有一点需要注意，那就是能够在某个绕路中进入另一个绕路。如果绕道通过“Start a Business”，并且生意已经破产，那么你就临时离开“Start a Business”，然后绕道通过“Messy Divorce”。一旦离开“Messy Divorce”，就返回到“Start a Business”方格中你离开时的原来位置。最后，你也离开“Start a Business”方格，回到主路中你开始绕路时的原来位置。在游戏板的其他不同地方也可能会采用同样的绕路(例如，Start a Business)。

与大多数棋盘游戏不同，Big Bux 不是必须得结束。你可以永远沿着这个游戏板一圈又一圈地行走。有以下三种方式可以结束该游戏。

- **退休**：为了做到这一点，必须拥有某一特定级别的资产，并且做出退休决定。
- **破产**：一旦失去所有资产，继续这个游戏就没有意义了。搬到皮若亚市忍辱偷生吧。
- **进监狱**：这是判断错误的后果，而且不是一种正常退出游戏的方式。

计算机程序也跟这个类似。当完成了打算做的所有事情之后，你可以选择退出程序，虽然如果想的话，你仍然可以继续运行该程序。如果已经完成文档或电子表格，可以保存它们并退出程序。反之，如果正在编辑一张照片，但是它看起来比以前还要糟糕，而且每次选择“锐化”之后都变得更加糟糕，那么你可以什么都不做就退出这个图形处理程序。如果你制造了一系列错误，那么这个程序将抛出错误信息，而且会破坏正在处理中的数据，这次程序运行以失败而告终。

再次强调一下，这是一个隐喻。不要按照字面意义去理解这个游戏板(在过去的 20 世纪 90 年代，硅谷的生活与之非常相像，我听说现在那里的生活已经平静多了)。

1.4 像棋盘游戏一样的汇编语言编程

我们已经根据棋盘游戏进行思考，接下来看一下图 1-2。这里所画的内容实际上与我在 25 或 30 年前在某些更简单的计算机上使用的汇编语言非常相似。那个标记为“程序指令”的列就是环绕在游戏板边缘的主路，这里只显示了游戏板的部分内容。这就是汇编语言程序，当执行时，那些真实的步骤和测试序列使计算机做某些有用的事情。汇编语言编程所做的事情实际上就是创建这样的程序指令序列。

其他内容都是临时的，它们会在(为正在进行中的游戏提供服务的)游戏板中间结束使命。大多数这些东西是包含了数据的存储位置。你也许已经注意到，这里涉及很多数据。(它们也是一些神秘的数字，例如，“004B”是什么意思？我将在第 2 章探讨这个主题。)我很抱歉，但是，游戏规则就是这样的。汇编语言就其本质而言，无非是一些数字而已，如果像某些人讨厌沙丁鱼一样讨厌数字，那么你就要度过一段苦难岁月了。(我喜欢沙丁鱼，它是我的传奇中的一部分。学会喜欢数字，它们并不是那么“咸”。)高级编程语言(如 Pascal 或 Python)通过将数字当成符号来处理来对其进行“化妆”——但是汇编语言，唉，就是你和数字。

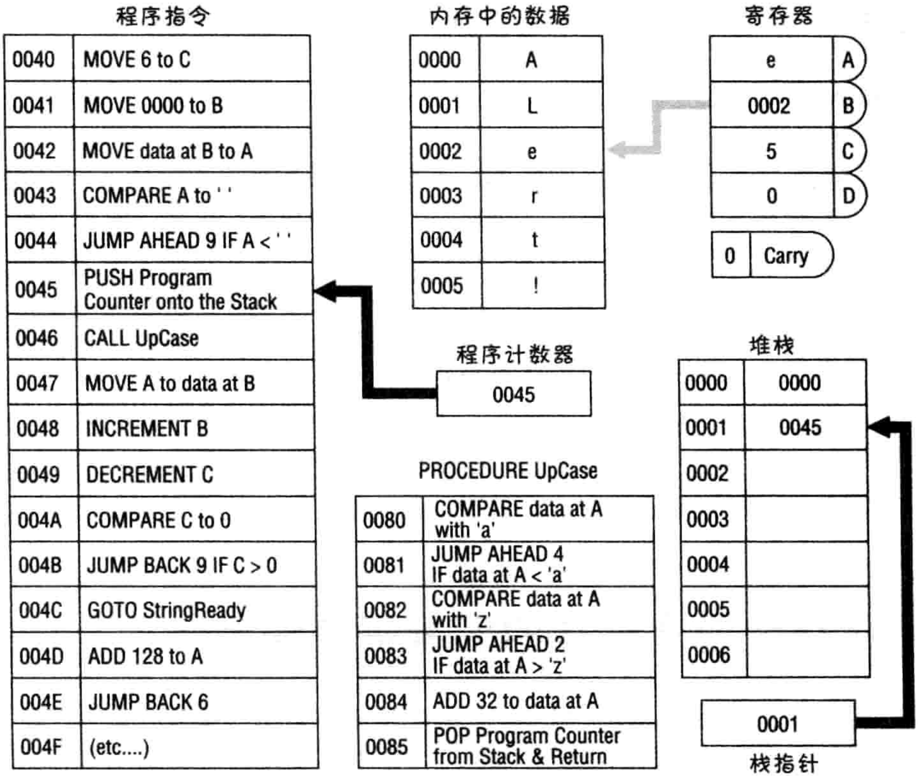


图 1-2 汇编语言游戏

注意，“汇编语言游戏”代表的并不是真正的计算机处理器，如奔腾处理器。而且，我已经把指令的名字变得比 Intel 汇编语言中的指令名更加容易理解。在真正的编程中，指令名通常情况下是诸如 STOSB、DAA、INC、SBB 以及其他一些如果没有足够的解释就根本无法理解的神秘内容。我们正从侧面轻松地进入它，所以，在本章中，我不得不为某些内容裹上一点糖衣，以便能够清晰地描绘这个隐喻。

1.4.1 代码和数据

像大多数棋盘游戏(包括 Big Bux 游戏)一样，汇编语言这个“棋盘游戏”包含两大类元素：游戏步骤和存放东西的位置。“游戏步骤”就是前面曾经提及的步骤和测试。存放东西的位置就是可以存放数据的小孔，要确保这些数据仍然存

放在初次存放它们的地方，直到把它们取出来，或者通过某种方法修改它们。

在编程术语中，那些游戏步骤叫作代码(code)，那些小孔中的数字(要与那些小孔本身区别开来)叫作数据(data)。那些小孔本身通常叫存储器(storage)。(区分存放信息的位置和存放在该位置的信息十分关键。不要把它们相混淆。)

Big Bux 这个游戏以相同的方式工作。回去再看看图 1-1，注意，在“Start a Business”这个绕路中，有一条指令：“Add $850,000 to checking account.”(添加 850,000 美元到支票账户。)支票账户就是 Big Bux 游戏的若干种不同存储器之一，而金额则是某种类型的数据。从概念上来讲，它与汇编语言游戏的读入“ADD 5 TO Register A” 指令没有任何不同。代码中的一条“ADD”指令改变了名为 Register A 的小孔中存放的数据值。

代码和数据是两种完全不同的东西，但是它们通过一些使游戏变得有趣的方式相互合作。代码包含一些把数据放入存储器(MOVE 指令)的步骤和一些修改存储器中已有数据的步骤(INCREMENT 和 DECREMENT 指令，还有 ADD 指令。)大多数时候，可以将代码看作数据的主人，因为是代码将数据写入存储器。另外，数据的确也影响代码。在代码进行的测试中，有一些需要检测存储器中的数据，如 COMPARE 指令。如果存储器中已有某一特定数据，代码做一件事情，如果存储器中没有该数据，代码做另外一件事情，就像在 Big Bux 游戏中的 JUMP BACK 和 JUMP AHEAD 指令一样。

标识为 PROCEDURE 的指令块是一条主程序流之外的绕路。在程序的任意位置都可以“潜入”这个过程，执行其中所包含的步骤和测试，然后再返回到离开时的那个位置。这样，那些通用的和使用频率较高的步骤和测试序列就可以只出现在一个地方，而不是在每个需要它们的地方都分别复制一份。

1.4.2 地址

在程序左侧表示步骤所在位置和数据所在位置的那些有趣的数字之中，还包含着另外一个重要的概念。每个数字都是独一无二的，因为那个数字所标识的位置只在电脑中出现一次。这个位置叫作地址(address)。在计算机中，数据是通过指定其地址来存储和检索的。而过程则是通过指定其起始地址来进行调用的。

标识为 PROGRAM COUNTER 的小方框 (也叫存储位置)保存着下一条即将执行的指令的位置。程序计数器里的数字递增 1(我们说，每当一条指令执行后程序计数器都递增 1，除非有指令告诉它做其他的事情)。以地址 004B 处的 JUMP BACK 9 指令为例。当这条指令被执行时，程序计数器将往上面跳回 9 个位置。

这与很多棋盘游戏中的“回跳 3 个方格”有着异曲同工之处。

1.4.3 隐喻，将军

眼下我能提供的关于汇编语言游戏的讲解就这么多。这里仍然是第 1 章，我们仍然处于隐喻阶段。略懂计算机的人大多能够识别和理解图 1-2 中所做的部分事情。(有一个真实且可追踪的程序在那里运行——希望你弄明白它所做的事情以及它是如何做这些事情的。)对计算机一窍不通的人们也不应因为完全迷惑不解而觉得落在后面。我编写汇编语言游戏的唯一目的是为了解释以下几点内容。

- **每一个单独的步骤都很简单** 一条单个指令所做的事情无非是把一个简单的字节从一个作为存储器的小孔中移到另一个中，执行非常基础的数学运算(如加法、减法)，或者把某个存储器小孔中的数值和存放在另一个存储器小孔中的数值进行比较。这是一个好消息。因为这样你就能够将精力集中于一条单个指令所执行的简单任务上，而不必被复杂性所淹没。
- **要想做一件有用的事情，需要好多步骤** 我们经常能够用五六行 Pascal 或者 BASIC 代码编写一个有用的程序。实际上，通过使用 Visual Basic 和 Delphi 等可视化编程系统，可以根本不用编写任何代码就创造有用的程序。(那些代码仍然在那里……但是，它们被密封起来，你只需在一个包含很多这样的块的集合中选择让哪个块中的封装代码来运行。)一个有用的汇编程序不会少于 50 行代码，一些更具挑战性的应用程序可能会达到上百行或者几千行，甚至上万行代码。汇编语言编程的技巧在于组织这几百条或者几千条指令，并使其仍然易于阅读和理解。
- **汇编语言的关键是理解内存地址** 在 Pascal 和 BASIC 这类语言中，编译器处理“内容存放在何处”之类的问题——你只需简单地为某些东西提供一个符号名，当需要查看或修改它的时候，随时通过那个名字来访问它。在汇编语言里，必须时刻知道哪些内容存放在计算机内存中的哪个位置。因此，在学习本书的过程中，对于内存寻址的概念要格外注意，其实，内存寻址无非是一种指出东西存放在何处的艺术。汇编语言游戏中充满了地址和使用这些地址进行工作的指令(例如 MOVE data at B to C 的含义是：把存放在由寄存器 B 指定的地址处的数据移动到寄存器 C 中)。内存寻址大概是汇编语言中最为棘手的部分，但是一旦掌握了它，你就已经掌握了汇编语言的真谛。

到目前为止，我讲的所有内容都是为了熟悉情况而准备的铺垫过程。我尽力让你大致了解一下汇编语言，以及它的基本原理如何与我们的日常生活相关联。生活就是一系列步骤和测试，棋盘游戏也是如此——汇编语言亦然。后面，当我们通过面对计算机数字的本质来继续探索汇编语言的真谛时，一定要牢记那些隐喻。

第 2 章

外星基数

拥抱二进制和十六进制

2.1　新数学怪物归来

那是 1966 年。也许你在那时已经出生了。新数学(New Math)突然“空降”全国小学课程，作业中开始包含数轴、集合和交替基数。中产阶级父母们为回答孩子的“17 以基数 5 表示是多少？”和“空应该隶属于哪个集合？”之类的问题而感到挠头。很快 (在我的记忆中，那是大约两个月的时间)，整个事情就被扔进了垃圾堆，就像它们一开始被那些无所事事的糊涂虫教育官员们引入时一样迅速。

实际上，这是一件憾事。那时，人们只是知道在小学中不再讲授老式数学了，但好像并没有意识到，或者承认，我们正在学习新数学——我们一直在想，知道松鼠集合和哺乳动物集合的交集会有什么用处。当然，真相是它没有任何用处。在美国，数学一直被当作应用数学即算术来讲授，重点在于应用题。如果你学了之后，依然不会平衡支票本或者计算处方比例，那么它就不是真正的数学。小学课堂很少或者几乎不讲解数学逻辑，部分是因为从历史角度来看，美国的小学已经成为一种为日常生活而开办的商业学校，获取少量的令人讨厌的基础文化修养已经足够困难了。让那些注重实际的中产阶级美国人充分意识到备用数制系统的美妙，已经超出了他们的许可范畴。

我属于那些对 1966 年得到支持的新时代风格数学十分感兴趣的少数人之一，但是当这种数学被淡忘时，我也乐得把它搁置一边。直到 1976 年，像手持绕线连接枪的狂躁者一样工作几个星期之后，当我把“火药”装入我的 COSMAC ELF 计算机，在 LED 显示器上得到一对基数为 16 的数字的问候时，我不得不又重新捡起它。

我的主呀，新数学又回来了……

之所以写这一章，是因为在汇编语言级别上，计算机并不理解我们日常生活中熟悉的以 10 为基数的数字。计算机以“有点精神分裂”的方式工作在基数 2 和基数 16 下——它总是这样。如果愿意继续使用高级语言，例如 C、BASIC、Pascal，可以忽略所有这些外星基数，或者将其当作已经熟练了解某种语言之后的其余高级主题。但是在这里不行。汇编语言的所有知识都取决于对这两种数字基数的彻底理解，所以，在开始其他事情之前，我们将再次学习如何计数——这次是在火星上。

2.2 在火星上计数

火星上存在着智慧生命。

就是说，火星人拥有足够的智慧，它们这 60 年来通过观看我们的电视节目了解到，兴旺的旅游业并不是它们的强项。所以它们仍然处于隐藏状态，只是偶尔简单地露个头，把大岩石雕刻成 Elvis 的面孔，以此帮助八卦小报《国家询问者》确保没有人再拿火星人当真。 火星人有时的确会与我这样的科幻作家通信，它们非常清楚没有人曾经认真对待过它们。所以，这一小节将包含一些关于火星人计数方式的内容。

火星人的一只手上有 3 个手指，另一只手上只有 1 个手指。男性火星人的左手长着 3 个手指，而女性火星人右手长着 3 个手指。这样，跳华尔兹舞和做某些其他事情就会更容易些。

像人类和其他智慧生命一样，火星人是通过数它们的手指来开始计数的。就像我们使用 10 根手指来把东西进行分组，而且把 10 作为基数一样，火星人使用 4 根手指来把东西进行分组，而且把 4 作为基数。随着时间的推移，我们的文明把一组由 10 个数字组成的集合作为了数制系统标准。类似地，火星人把一组由 4 个数字组成的集合作为了它们的数制系统标准。这 4 个数字以及它们按照火星人发音的名字如下： Θ (xip)，∫ (foo)，∩ (bar)，≡ (bas)。

像我们的 0 一样，xip 是一个占位符，表示没有内容，尽管火星人有时从 0 开始计数，但是它们更常用的是从 foo 开始计数，foo 代表一个东西。所以火星人这样开始计数：foo,bar,bas……

现在怎么办？bas 之后的计数该如何处理？表 2-1 演示了外星人如何数到我们人类所说的 25。

表 2-1 火星计数，以 fooby 为基数

火星数字	火星发音	地球上与之相等的数
Θ	Xip	0
⌠	Foo	1
∩	Bar	2
≡	Bas	3
⌠Θ	Fooby	4
⌠⌠	Fooby-foo	5
⌠∩	Fooby-bar	6
⌠≡	Fooby-bas	7
∩Θ	Barby	8
∩⌠	Barby- foo	9
∩∩	Barby- bar	10
∩≡	Barby- bas	11
≡Θ	Basby	12
≡⌠	Basby- foo	13
≡∩	Basby- bar	14
≡≡	Basby- bas	15
⌠ΘΘ	Foobity	16
⌠Θ⌠	Foobity- foo	17
⌠Θ∩	Foobity- bar	18
⌠Θ≡	Foobity- bas	19
⌠⌠Θ	Foobity-fooby	20
⌠⌠⌠	Foobity-fooby-foo	21
⌠⌠∩	Foobity-fooby- bar	22
⌠⌠≡	Foobity-fooby- bas	23
⌠∩Θ	Foobity-barby	24
⌠∩⌠	Foobity-barby-foo	25

只有 4 个数字(包括一个代表 0 的数字)，如果不超出这几个数字范围，火星人只能数到 bas。bas 后的数字有一个新的名字：fooby。 fooby 是火星数制系统的基数，而且也许是火星数制系统中最重要的数字。fooby 是火星人拥有的手指个数。我们称之为 4。

关于 fooby，最重要的事情是火星人将其以数字形式写出的方式：ʃΘ。火星人不是用简单的一列而是用两列来表示 fooby。像我们人类的十进制数制系统一样，每一列都是 fooby 的幂。这里所包含的意思是：从右往左看，每一列代表的(权)值等于 fooby 乘以其右侧列的(权)值。

最右侧的列代表的是以 foo 为单位的个位；下一列代表的是 fooby 乘以 foo，或者(已知火星算术和我们人类这里的一样，还是新数学)简写为 fooby。fooby 左侧的那一列代表 fooby 乘以 fooby 或者简写为 foobity，以此类推。这种关系用表 2-2 表示应该更清晰些。

表 2-2　fooby 的幂数

ʃ	Foo	×Fooby= ʃΘ	(Fooby)
ʃΘ	Fooby	×Fooby= ʃΘΘ	(Foobity)
ʃΘΘ	Foobity	×Fooby= ʃΘΘΘ	(Foobidity)
ʃΘΘΘ	Foobidity	×Fooby= ʃΘΘΘΘ	(Foobididity)
ʃΘΘΘΘ	Foobididity	×Fooby= ʃΘΘΘΘΘ	(Foobidididity)
ʃΘΘΘΘΘ	Foobidididity	×Fooby= ʃΘΘΘΘΘΘ	以此类推

2.2.1　火星数字剖析

任一给定列都可以包含一个从 xip(0)到 bas(3)之间的数字(digit)，以指出整个数值(number)中包含多少个该列(权)值。让我们通过一个例子进行说明。请看图 2-1，它是对于火星数字∩≡ʃΘ≡的剖析，该数字的发音是“Barbididity-basbidity-foobity-bas”。

最右侧的列表明这个数值中包含多少个“个位列”的权值。那里的数字是 bas，表明这个数字包含 bas 个单位值。从右数第二列的权值等于 fooby 乘以 foo 的值(fooby*1)，或者 fooby。fooby 列的 xip 表示这个数字里没有 fooby。ʃΘ中的 xip 是一个占位符，就像我们人类数制系统中的 0 一样。还要注意，在如下所示的数

字矩阵右侧的分列和中，表示 fooby 列的值的这一行用两个 xip 表示。不仅有一个 xip 用于表明没有 fooby，而且有一个 xip 出现在 foo 列的位置上。当我们向更重要的左侧列移动时，这种模式继续在该分列和中延伸。

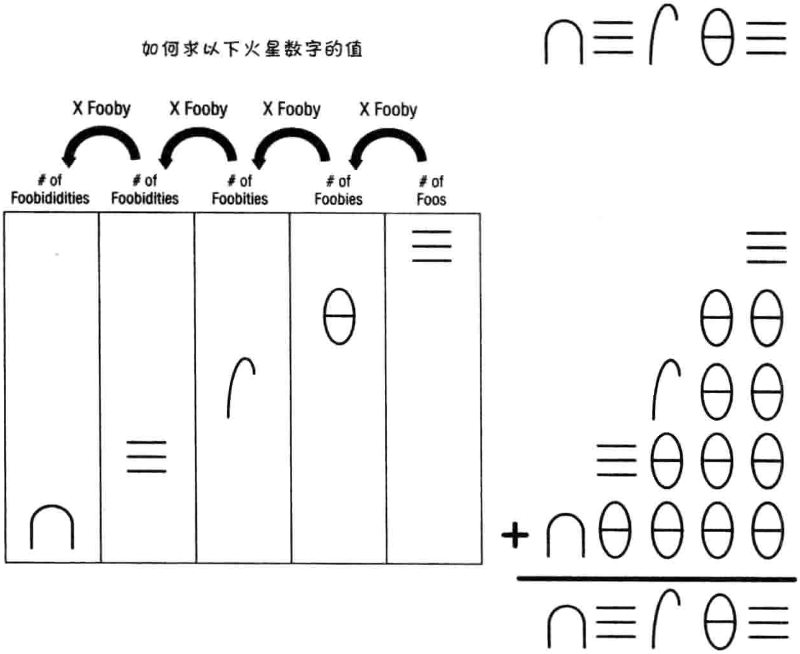


图 2-1　剖析∩≡⌠Θ≡

fooby × fooby=foobity。⌠告诉我们，这个数字有 foo 个 foobity(1 个 foobity)。下一列，继续延续这种模式，是 foobity×fooby，或者是 foobidity。在分列表示法里，foobidity 被写成⌠Θ Θ Θ。≡告诉我们，这个数有 bas(3)个 foobidity。basbas foobidity 也拥有单独的名字，即 basbidity，它可被写成≡Θ Θ Θ。请注意一下分列和中 basbidity 的出现。

再左一列的(权)值为 fooby×foobidity，或者 foobididity。∩告诉我们，此数字有 bar(2)个 foobididity。Barbar foobididity (写成∩Θ Θ Θ)也拥有单独名字，即 barbididity。也请注意一下分列和中 barbididity 的出现以及 4 个用于保留 4 个空列位置的数字 xip。

这个分列和陈述了一种数字组装方式：该数字包含 barbididity、basbidity、foobity 和 bas。通过一个简单地加法，把所有这些数字“卷起来”，就会得到∩≡

ʃΘ≡。其名字可以通过简单地把这些组成部分的值连接起来而实现：barbididity-basbidity-foobity-bas。注意，名字里没有代表空 fooby 列的部分。例如，在人类熟悉的十进制数制系统里，我们不把数字 401 读成四百零一。同样，火星人并不读“xip fooby”，而是省略它。

作为一个小练习，请根据目前讲授的关于火星数制系统的知识，推算出与∩≡ʃΘ≡相等的地球数字。

2.2.2 数字基数的本质

既然近期内根本不可能到火星上去旅行，那么我们地球人了解火星数制系统有什么用处呢？仅仅出于这个原因：要想了解某种数字基数而不必受我们熟悉的数字及常用基数 10 的干扰，它是一种绝妙的方法。

在多列式数字表示法中，例如我们人类和火星人使用的，数制系统的*基数*是一个量值，通过这个量值，数字的每一列可以超出它右侧列的值。在我们的十进制系统中，每一列代表它右侧列的权值与 10 相乘的结果。在一个像火星上使用的 fooby 基数系统中，每一列都代表它右侧列的权值与 fooby 相乘的结果。(假使你还没有理解的话，火星人实际上在使用基数 4——但是我想让你从火星人的角度来理解它。)每一个数制系统都有一套数字符号，这些数字符号的个数与其基数相同。在我们的 10 基数数制系统中，有从 0 到 9 共 10 个数字符号。在火星的 4 基数数制系统中，有从 0 到 3 共 4 个数字符号。在任意给定的基数数制系统中，基数本身都不能用一个简单的单个数字来表示！

2.3 八进制：绿色精怪怎样偷走 8 和 9 的

让我们告别火星。因为除了大量的氧化铁和一些看起来花里胡哨的地貌之外，火星并没有更多的东西可提供给我们 10 个手指的人类。这里也使用一些奇数基数，我想采取绕道的方式，先介绍一个公司，它几乎占领了地球上的一个单独的世界：数字设备公司，它更以 DEC 这个名字而闻名于世。

回到 20 世纪 60 年代。DEC 发明了小型机，并将其作为 IBM 创造的硕大而昂贵的大型机的挑战者。(小型机的时代已经故去很久了，而 DEC 本身也已经成为历史。)为了确保没有软件可以从 IBM 大型机中移植到 DEC 的小型机上，DEC 将它的机器设计为只能理解以 8 为基数的数制系统。

让我们根据火星经历思考一下。在以 8 为基数的数制系统中，肯定有 8 个数字。DEC 考虑得非常周到，它并没有创造自己的数字符号，所以，它使用的是我们传统地球人使用的数字 0 到 7。在以 8 为基数的数制系统中，并没有数字 8！这常常会需要一个习惯的过程，但是它是一个数字基数定义的一部分。DEC 将它的这个以 8 为基数的数制系统命名为：八进制。

在八进制中，分列数字所遵循的规则同我们思考火星数制系统时遇到的规则类似：每一列都有一个权值，该权值等于基数乘以其右侧列的权值。(最右侧的列就是单位列。)就八进制而言，每一列的权值都等于 8 乘以它右侧列的权值。

谁偷走了 8 和 9

关于八进制，演示要比讲解更容易理解一些。八进制以我们非常熟悉的方式进行计数：1，2，3，4，5，6，7……10。

这就是麻烦开始的地方。在八进制中，10 在 7 的后面。8 和 9 发生了什么事情呢？是绿色精怪(还是火星人)刚刚偷走了它们吗？它们仍然在那里——但是拥有了不同的名字。在八进制里，当你说“10”的时候，你的意思是“8”。更糟糕的是，当你说“11”的时候，你的意思是“9”。

非常不幸的是，DEC 并没有为这些列值发明一个机灵的名字。当然，第一列是单位列。单位列左侧的这一列是十位列，就像我们的十进制数制系统一样——但是，这就是难点所在，也是我把火星人拉到这里来的原因：八进制里的“十位”列实际上拥有的权值是 8。

计数表可能会对理解有所帮助。表 2-3 计数到八进制的 30，即十进制的 24。除了以 10 为基数的数制系统之外，我不喜欢使用十一、十二等这样的术语。但是，八进制中的习惯经常要把这些数字像在十进制里一样的发音，只是在它们后面加上了“八进制”这个词。不要忘记说八进制，否则会真正把人们弄糊涂的。

表 2-3 八进制计数，以 8 为基数

八进制数	八进制发音	与之等值的十进制数
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4

续表

八进制数	八进制发音	与之等值的十进制数
5	5	5
6	6	6
7	7	7
10	10	8
11	11	9
12	12	10
13	13	11
14	14	12
15	15	13
16	16	14
17	17	15
20	20	16
21	21	17
22	22	18
23	23	19
24	24	20
25	25	21
26	26	22
27	27	23
30	30	24

记住，在任意一个基于给定数字基数的数制系统中，每一列的权值都等于基数值乘以它右侧列权值，所以，在八进制中，“十位”列实际上是八位列。(他们称之为十位列，因为它被写作 10，而且发音也是“十”。)类似的，十位列左侧的这一列是百位列 (因为它被写作“100”，而且发音也是“百”)。但是，百位列的权值实际上是 8 乘以 8，或者 64。百位列左侧的列权值为 8 乘以 64，或 512。再左侧的列权值为 8 乘以 512，或者 4096。

这就是为什么当某些人谈到 “八进制 10”的值时，他们的意思是 8；当谈到“八进制 100”的值时，他们的意思是 64；以此类推。表 2-4 总结了八进制列权值和与之相等的十进制值。

表 2-4　作为 8 的幂数的八进制列

八进制数	8 的幂数	相当于	十进制数	八进制数
1	$=8^0$	=	1×8	= 10
10	$=8^1$	=	8×8	= 100
100	$=8^2$	=	64×8	= 1000
1000	$=8^3$	=	512×8	= 10000
10000	$=8^4$	=	4096×8	= 100000
100000	$=8^5$	=	32768×8	= 1000000
1000000	$=8^6$	=	262144×8	= 10000000

在第一列中的数字(单位，或个位列)表明在这个八进制数中包含多少个单位量。左侧的下一列(即十位列)的数值表明这个八进制数中包含多少个 8。第三列(即百位列)中的数字表明这个八进制数中包含多少个 64，等等。例如，八进制 400 的意思是这个数包含 4 个 64，用十进制表示是 256。

是的，它很容易让人感到困惑，肯定会这样。理解它的最好方式是剖析一个普通的八进制数，就像我们剖析一个普通的火星数字一样。这就是图 2-2 所做的事情：我们将八进制数字 76225 拆成列，然后再将这些列加起来。

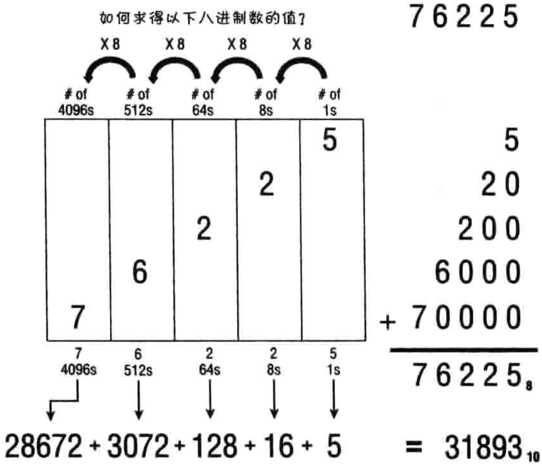


图 2-2　剖析八进制数 76225

这里的工作方式同火星上的、十进制的或者任意你能想出的其他数字基数的工作方式类似。用一般的(有点正式的)术语来讲：每一列都有一个权值，该权值

等于数字基数的“列位置序号减 1”次幂。例如，第一列的权值是数字基数的 1 减 1 次幂，或者零次幂。由于任意数字的零次幂都是 1，所以在以任意数字为基数的数制系统中，第一列的值总是 1，并且通常被称为单位列(units column)。第二列的权值是数字基数的 2 减 1 次幂，即这个数字基数本身。在八进制中就是 8。在十进制中就是 10；在火星的以 fooby 为基数的数制系统中就是 fooby。第三列的权值是数字基数的 3 减 1 次幂，或 2 次幂，以此类推。

每一列上的数字代表这个数包含了多少个该列的权值。这里，8 进制数 76225 中的 6 告诉我们，在整个八进制数 76225 中有 6 个这列权值的实例。6 占的是第 4 列，第 4 列的权值为 $8^{(4-1)}$，即 8^3，或 512。这告诉我们，在整个数中有 6 个 512。

通过将外星(非十进制)基数的每一列的权值乘以该列上的数字(来求取与之相等的十进制数)，然后再将这些数字相加，我们可以把任意基数的数字转换成十进制(以 10 为基数的数制系统)数字。这些都在图 2-2 中进行了很好的阐释。八进制数和与之等值的十进制数并排放在一起。注意图 2-2 中分列和右侧的下标数字。这些下标在许多技术出版物中用于指示数字基数。例如，76225_8 的下标指明了这里的 76225 是一个八进制值，它的基数是 8。不像火星数字和我们传统的十进制系统数字之间的区别那么明显，对于八进制本身，的确没有什么使之称其为八进制的特殊之处。(在稍后的十六进制知识讲解中，我们将会遇到某些类似的问题。) 相应地，数值 31893_{10} 通过下标表明它是一个以 10 为基数的数，或十进制数。这种书写方式大多数情况下出现在科技作品中。在大多数计算机类出版物中(包括本书)也使用一些其他的表示方法，本书后续部分会有更多与之相关的介绍。

我们不仅从火星数制系统的角度，而且从人类十进制数制系统的角度研究了分列表示法，在继续进行下一步的学习内容之前，确保你已经理解了这些分列表示法在任意基数数制系统下的工作原理。

2.4　十六进制：解决数字的短缺

八进制对你来说不可能有任何用处，除非你像我的朋友一样，恢复一台古老的 DEC PDP8 计算机，那是他从就读的大学按淘汰资产购买的，以镑计价(据他说比土豆便宜得多，只是不像土豆那么容易煎炸)。像前面提到的那样，在微机世界里，真正面对的数制系统是以 16 为基数的，我们称之为十六进制(hexadecimal)，或者(更亲切地)简单的“hex”。

十六进制享有所有数字基数的本质属性，包括火星上的和八进制的：它是一种分列表示方法，在这种分列表示法中，每一列的权值都等于 16 乘以它右侧列的权值，从 0 开始，一直到多少？

这里，我们的数字不够用了。从 0 到 9，我们很容易写出来(因为它们本来就是单个字符)。然而，10、11、12、13、14 和 15 需要用某种类型的单个符号来表示。20 世纪 50 年代早期，十六进制表示法的发明者们没有增加任何数字，而是借用了字母表中的前 6 个字母，将其作为所需的数字。

那么，在十六进制下计数是这样来进行的：1、2、3、4、5、6、7、8、9、A、B、C、D、E、F、10、11、12、13、14、15、16、17、18、19、1A、1B、1C，等等。表 2-5 以一种更有条理的方式，重述了这些内容，同时附上了与之等值的 0～32 之间的十进制数。

表 2-5 以十六进制计数，以 16 为基数

十六进制数	发音(后面跟“HEX”)	十进制数表示
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
A	A	10
B	B	11
C	C	12
D	D	13
E	E	14
F	F	15
10	Ten (或 One-oh)	16
11	One-one	17

续表

十六进制数	发音(后面跟“HEX”)	十进制数表示
12	One-two	18
13	One-three	19
14	One-four	20
15	One-five	21
16	One-six	22
17	One-seven	23
18	One-eight	24
19	One-nine	25
1A	One-A	26
1B	One-B	27
1C	One-C	28
1D	One-D	29
1E	One-E	30
1F	One-F	31
20	Twenty (或 Two-oh)	32

我喜欢的其中一个十六进制习惯是丢掉诸如 11 和 12 之类的词，它们和我们的十进制绑定得太紧，只会增加令人不快的混乱。面对十六进制的 11(通常写成 11H 来指明我们谈论的基数是多少)，我们将其读成“十六进制一一”。再次强调，不要忘记在一个十六进制数的后面加上“hex”，以避免令人不快的混乱。但是对于数字 0 到 9，由于它们的值在十六进制中和在十进制中是相同的，因此就不必再加“hex”了。

有些人仍然喜欢像“十六进制十二”这样来读十六进制数，这样读是有效的，它的意思是十进制 18；但是我不喜欢这种读法，建议摒弃它。在没有佩戴查理·卓别林的面具之前，单是外星基数业务本身就足够令人困惑了。

十六进制的每一列的权值都等于 16 乘以其右侧列的权值。(最右侧的列，像所有基数中的一样，是单位列，它的权值是 1。)正如你猜想的那样，当从右向左移动时，每一列的权值的增长速度快得可怕。表 2-6 展示了十六进制中的前七列。出于比较的目的，请注意十进制表示法中的第七列权值为 100 万，而在十六

进制下，第七列的权值为 16 777 216。

表 2-6　权值为 16 的幂数的十六进制列

十六进制	16 的幂数	十进制
1H	$=16^0=$	1 × 16 = 10H
10H	$=16^1=$	16 × 16 = 100H
100H	$=16^2=$	256 × 16 = 1000H
1000H	$=16^3=$	4096 × 16 = 10000H
10000H	$=16^4=$	65 536 × 16 = 100000H
100000H	$=16^5=$	1 048 576 × 16 = 1000000H
1000000H	$=16^6=$	16 777 216……

为了帮助大家理解十六进制数的组成原理，我已经在图 2-3 中剖析了一个中等大小的十六进制数，表述方式跟我在前面剖析火星基数 fooby 和八进制中的基数 8 类似。像在八进制中一样，0 只是在一列中占个位置，不会往整个数上增加任何值。注意，在图 2-3 中的八进制数 3C0A9H 中，有 0 个 256，即没有 256。

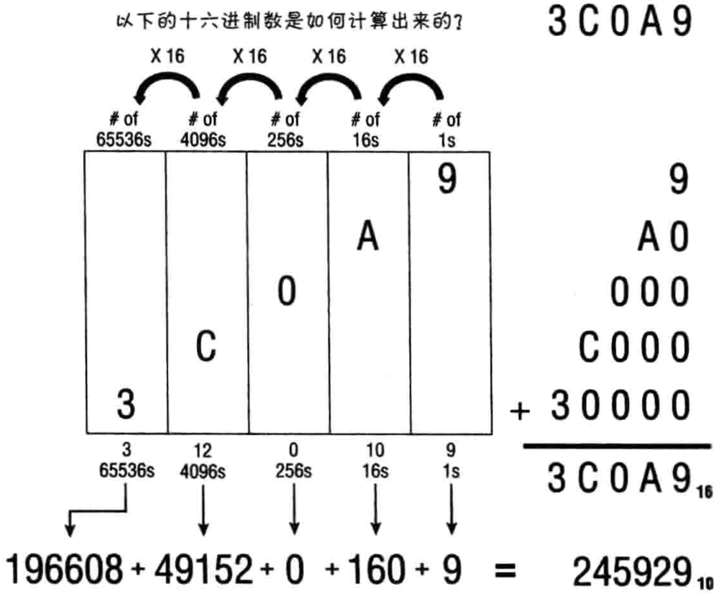


图 2-3　剖析十六进制数 3C0A9H

像图 2-2 中所示的那样，每一列的权值的十进制值都在该列的下方，所有列的和都以十进制和十六进制两种方式进行表示。(注意下标！)

2.5　从十六进制到十进制，从十进制到十六进制

我们执行的大多数十六进制操作是简单的十六进制和十进制之间的双向转换。执行这类转换的最简单方法是使用一个十六进制计数器，或者是一个真正的计算器，例如：古老的 TI Programmer(我至今还有一个，它靠电池来工作)，或者是一个支持十六进制的软件计算器。在 Ubuntu Linux 自带的计算器中，如果选择“查看”→“科学”菜单项，可以进行十进制、十六进制和二进制数学计算。Windows 计算器的工作方式相同：默认视图是基础视图，必须选择“科学”视图来进行任何其他基数的运算。随 Kubuntu/KDE 4 默认安装的 SpeedCrunch 计算器一开始就能够处理十六进制和二进制。

使用计算器并不需要你任何脑灰质，当然，也不会对你理解十六进制系统带来更多的帮助。所以，当你对这些外星基数还是一个相对新手时，“解雇”一切理解十六进制的东西，管它是硬件、软件还是人类伙伴。

实际上，当你正处于学习阶段时，最好的工具是一个简单的四功能存储器计算器。这里讨论的转换方式都使用这样一个计算器，我正在努力教给你的是数字基数转换，而不是十进制加法或者长除法。

2.5.1　从十六进制到十进制

你会逐渐发现，把十六进制数转换成十进制数比其他操作容易得多。要想完成图 2-1、图 2-2 和图 2-3 所示的数字剖析效果，一般的方法是：获取十六进制数的每一列代表的十进制值，然后再将所有这些值相加。

让我们通过一个简单的十六进制数来试验一下。以十六进制数 7A2 为例。从最右侧的列开始。这是所有基数数制系统中的单位列。这里有 2 个单位，所以，在计算器中输入 2。现在，把 2 存入内存(或者按一下 SUM 按钮，如果有该按钮的话)。

对于单位列，所做的事情就这么多。请记住，你真正在做的事情是对于该十六进制数的每一列的值进行累计。往左移到下一列。记住，在十六进制中，每一列的权值都等于 16 乘以它右侧列的权值，所以第二列的权值是 16。(如果你记不清每一列的权值，可以参考表 2-6。)在第二列里，有一个数字 A。在十六进制

中，A 代表十进制的 10。所以，整个这个列的值为：16*10，或者 160。在计算器上执行这个乘法，然后将结果与存放在内存中的 2 相加。(再次强调一下，如果你的计算器上有 SUM 按钮，可以通过这个按钮来完成这件事。)

记住你正在做的事情是：求出每一列的十进制值，然后进行累计。现在移到从右边数第三列。这列有个数字 7。第三列的权值是 16*16，或者 256。用计算器实现 256*7，并将结果加到已经求得的累计值上。

任务完成。从计算器内存中取出最终的累计值(即计算结果)。结果为 1954，它就是与 7A2H 等值的十进制数。

好了！让我们采用更少的讲解和一个更大的十六进制数 C6F0DBH 来做个更快速的试验。

(1) 计算单位列。 B×1 = 11×1 = 11。此时，累计值为 11。
(2) 计算权值为 16 的列。D×16 = 13×16 = 208。把 208 加到累计值上。
(3) 计算权值为 256 的列。 0×256 = 0。 继续进行下一步。
(4) 计算权值为 4 096 的列。F×4 096 =15×4 096 = 61 440。把它加到累计值上。
(5) 计算权值为 65 536 的列。6×65 536 = 393 216。把它加到累计值上。
(6) 计算权值为 1 048 576 的列。C×1 048 576=12×1 048 576=12 582 912。把它加到累计值上。

最终累计值即运行结果应该是 13 037 787。

最后，请你在没有任何帮助的情况下，自己计算一下十六进制数 1A55BEH 的十进制值。

2.5.2 从十进制到十六进制

此时，你应该已经理解十六进制了。这很好，因为如果要进行逆向转换，即从我们熟悉的十进制到十六进制，要更加困难些，而且包含的数学运算更多。我们必须找出在这个十进制数中包含多少个各个十六进制列权值——而且其中包含相当多令小学五年级学生望而生畏的运算——长除法。

让我们开始做这件事情，也是从一个非常简单的十进制数开始。以 449 为例。计算器开始“复仇”了。输入数字 449 并且将其存入计算器的内存。

我们需要做的事情是找出包含在十进制数 449 中的最大的十六进制列上的值，至少一次。还记得小学里的整除运算吗？(12 整除 855 多少次？)同学们经

常使用除法来找出一个数被包含在另一个数里多少次。这里也有点类似。回过头来看表 2-6，我们可以看到，对于十进制数字 449 而言，256 是最大的 16 的幂数，因此，最大的 16 进制列的值，至少被 449 包含一次(下一个最大的 16 的幂数——512——很明显，太大了，不可能被 449 所包含)。

因此，我们从 256 开始，判断一下 256 整除 449 多少次：449 / 256 = 1.7539。至少一次，但是不到两次，所以 449 只包含一个 256。在纸上写下数字 1。不要把它输入计算器。这里我们不再做累加了；更可能的是，你可以说我们正在对余数“记流水账”。“1”是与 449 相等的 16 进制数的最左侧列的值。

我们知道，只有一个 256 包含在 449 中。现在我们必须做的事情是从初始值中减掉 256，我们已经通过把 1 写在纸上完成了对 256 的“计数”。从 449 中减掉 256。把结果 193 存入内存。

我们已经从正在转换的数字中移除了权值为 256 的列。现在我们开始求取它右侧的列的值，即权值为 16 的列上的数字。193 里包含了多少个 16 呢？193 / 16 = 12.0625。它的意思是与 449 等值的十六进制数的权值为 16 的列值为……12？嗯。记住，要把这个数字存储起来，而且在十六进制里，我们把值为 12 的数用字母 C 表示。从十六进制的角度来看，我们已经找出，原始值在权值为 16 的列上包含了数字 C。把 C 写到纸上，放在刚才写的 1 的右侧：1C。到目前为止，一切都棒极了。

我们已经得到了权值为 16 的列上的数字，像找到权值为 256 的列上的数字一样，我们必须把它从剩余原始值(下面我们将其简称为余数)中减掉。权值为 16 的列的整个值为：C × 16 = 12 × 16 = 192。从计算器的存储器里取出 193，将其减去 192。只剩下一个孤零零的 1 了。

所以我们继续看一下单位列的值。在第一列上只有一个单位值，很明显，权值为 1 的单位列上，只有数字 1。把 1 写在刚才求取的 16 进制数中的 C 的右侧：1C1。与十进制数 449 相等的十六进制数是 1C1。

现在，也许你充分意识到为什么程序员那么喜爱十六进制计算器了。

回顾一下十进制—十六进制转换的“大图”。我们先找出藏在十进制数字中的、最大的十六进制列，求出那个列代表的值，再将那个值从十进制数中减掉。然后我们再寻找下一个比较小的十六进制列，然后再找出下一个更小的列……以此类推，在我们进行寻找的过程中，把每一列代表的值从十进制数字中减掉。从某种意义上讲，我们在把这个数字用连续的、依次减小的 16 的幂数除，当记录结果时，通过把每一列的值减掉来求取连续的余数。

让我们再试一次。这次要处理的数字是 988 664。

(1) 从表 2-6 中找出包含在 988 664 中的最大的十六进制列：65 536。988 664/65 536 = 15 和余数。先忽略余数。在十六进制中，15 = F。把 F 记下来。
(2) 从 988 664 中减掉 F × 65 536。存储余数：5 624。
(3) 继续寻找下一个较小的十六进制列。5 624 / 4 096 = 1 和余数。把 1 记下来。
(4) 从余数中减掉 1 × 4 096：5 624 − 4 096 = 1 528。存储余数：1 528。
(5) 继续寻找下一个更小的列。1 528 / 256 = 5 和余数。把 5 记下来。
(6) 从存储的余数 1 528 中减掉 5 × 256，把 248 存起来，作为新的余数。
(7) 继续寻找下一个更小的列。248 / 16 = 15 和余数。在十六进制中，15=F。把 F 记下来。
(8) 从存储的余数 248 中减掉 F × 16。余数 8 是最后的单位列上的值。把 8 记下来。

这样，就得到了右侧的结果：十进制 988 664= F15F8H。

注意十六进制数字后面的字母 H。从现在开始，本书中的所有十六进制数的末尾都会附加一个 H。这非常重要，因为并不是每个十六进制数字都包含字母数字，尖叫着说出这个数字的基数是 16 的事实。有 157H，也有十进制 157，但是二者并不代表同一个数。(现在，请快速说出：它们之间相差多少？)在编写汇编程序时，不要忘记写 H，这一点我将会在后面的内容中继续提醒你。

2.5.3 练习！练习！再练习

要充分了解十六进制表示法，最佳(实际上，也是唯一的)方法就是多用它。把下列每一个十六进制数都转换成相应的十进制数。把每一个数展示在剖析表里，指出这个数里包含多少个 1，多少个 16，多少个 256，多少个 4,096，等等，然后在将其加起来，构成相应的十进制数：

```
CCH
157H
D8H
BB29H
7AH
8177H
A011H
```

```
99H
2B36H
FACEH
8DB3H
9H
```

完成之后，现在倒过来，把下面的每一个十进制数都转换成相应的十六进制数。请记住一般方法：根据表 2-6，选出比这个要转化的十进制数小的最大的 16 的幂数。找出该十进制数里包含了多少个该 16 的幂数，然后把它作为被转换的十进制数的最左侧的十六进制数字写下来。然后，再从这个十进制数中减去该十六进制数字代表的整个值。通过使用下一个较小的 16 的幂数，重复这个过程，直到已经把这个十进制数减到 0 为止。

```
39
413
22
67,349
6,992
41
1,117
44,919
12,331
124,217
91,198
307
112,374,777
```

(为最后一个额外加分。) 如果你需要更多的练习，请选择一些十进制数，然后将其转换成十六进制，再从十六进制转换回来。做完之后，可以通过手头的任意一款能够进行十六进制运算的计算器来进行检验。

2.6 十六进制运算

对汇编语言越来越熟悉后，你将会进行越来越多的以 16 为基数的运算。你甚至(天哪)开始心算。当然，这仍然需要更多的实践。

十六进制加法和减法基本上和在十进制中的一样，只是有一些额外的数字作为特色添加进来。诀窍就是知道一直到 0FH 的加法表。这是不用亲自思考的最好办法。“如果 C 是 12，F 是 15，那么 C + F 是 12 + 15，也就是十进制的 27，十六

进制的 1BH。”取而代之的是，你应该只想到“C + F 是 1BH”。

是的，类似的提问次数太多了，但是我现在问你，并且在你今后的汇编之旅中，我还会一如既往地问你同一个问题，你是想驾驭汇编还是只是想混日子？学习钢琴需要练习，学习驾驭汇编语言编程的核心技术同样需要练习。

所以，让我像个年长的女教师一样，要求你记住以下内容。必要时你也可以制作抽认卡。

9	8	7	6	5			
+1	+2	+3	+4	+5			
0AH	0AH	0AH	0AH	0AH			
A	9	8	7	6			
+1	+2	+3	+4	+5			
0BH	0BH	0BH	0BH	0BH			
B	A	9	8	7	6		
+1	+2	+3	+4	+5	+6		
0CH	0CH	0CH	0CH	0CH	0CH		
C	B	A	9	8	7		
+1	+2	+3	+4	+5	+6		
0DH	0DH	0DH	0DH	0DH	0DH		
D	C	B	A	9	8	7	
+1	+2	+3	+4	+5	+6	+7	
0EH	0EH	0EH	0EH	0EH	0EH	0EH	
E	D	C	B	A	9	8	
+1	+2	+3	+4	+5	+6	+7	
0FH	0FH	0FH	0FH	0FH	0FH	0FH	
F	E	D	C	B	A	9	8
+1	+2	+3	+4	+5	+6	+7	+8
10H	10H	10H	10H	10H	10H	10H	10H
F	E	D	C	B	A	9	
+2	+3	+4	+5	+6	+7	+8	
11H	11H	11H	11H	11H	11H	11H	
F	E	D	C	B	A	9	
+3	+4	+5	+6	+7	+8	+9	
12H	12H	12H	12H	12H	12H	12H	
F	E	D	C	B	A		
+4	+5	+6	+7	+8	+9		
13H	13H	13H	13H	13H	13H		

F + 5 14H	E + 6 14H	D + 7 14H	C + 8 14H	B + 9 14H	A + A 14H
F + 6 15H	E + 7 15H	D + 8 15H	C + 9 15H	B + A 15H	
F + 7 16H	E + 8 16H	D + 9 16H	C + A 16H	B + B 16H	
F + 8 17H	E + 9 17H	D + A 17H	C + B 17H		
F + 9 18H	E + A 18H	D + B 18H	C + C 18H		
F + A 19H	E + B 19H	D + C 19H			
F + B 1AH	E + C 1AH	D + D 1AH			
F + C 1BH	E + D 1BH				
F + D 1CH	E + E 1CH				
F + E 1DH					
F + F 1EH					

如果没有其他内容的话，通过这个练习，你应该为计算机并不工作在以 64 为基数的数制系统下这一事实而感到庆幸。

2.6.1 列和进位

通过把所有的单列加法提交给内存(差不多是这样)，我们可以解决多列加法。它的工作原理与十进制加法原理完全类似。从右边开始，把每一列相加，然后每当某列的值超过 0FH 时，向下一列进位。例如：

```
  1     1
  2 F 3 1 A DH
+ 9 6 B A 0 7H
--------------
  C 5 E B B 4H
```

一列一列地认真完成这个加法。第一列的和(即最右侧的列) 是 14H，这个值一列放不下，所以我们必须向左侧的下一列进一位。第二列即使有了额外的进上来的 1，其和仅为 0BH，可以在一列中放下，所以不需要进位。

继续往左执行加法。从左数第二列也溢出了，需要向最后一列进一位。

只要记住单个数位之和，就没有问题。

嗯，差不多。

但是，这里有些内容应该留意。

两个数的单列相加，最多只能向上一列进一位。

到底使用以多少为基数的数制系统并不重要：16、10、fooby，或者 2。或者向上一列进一位，或者根本不进位。不知什么原因，很多人都对这一事实感到惊讶，所以请问一下自己：在我们熟悉的古老的十进制中，有哪两个数相加能够一下子向上一位进两位？最大的数是 9，9 + 9 = 18。记下 8，向上进一位。即使不得不加上一个从前一列进上来的进位值，你 (最多)得到 19。再次强调，要么进一位，要么不进位。当你在纸上或者在 CPU 的硅芯片上将两个数相加时，这一点非常重要，你会在若干章中体会到这一点。

2.6.2 减法和借位

如果你已经记住了单列相加，那么在做减法时，你通常会由于逆向思考而备受折磨。“如果 E + 6= 14H，那么 14H – E 必须 = 6” 。另外一种选择是记住一张更大数的表，因为我都还没有记住它们，所以不会要求你这样做。

但是久而久之，有些事情总会发生。在十六进制减法中，通过展示一个我们熟悉的十六进制加法，你应该能够预测出任意的单列减法；像在十进制数中一

样，多列减法也是一列一列地进行，每次完成一列的相减：

```
  F76CH
- A05BH
-------
  5711 H
```

在观察每一列的时候，你应该这样问自己："底下的数和什么数字相加能够得到上面的数？"这里，你应该从加法表中得到 B + 1 = C，所以 B、C 之间相差 1。最左侧列实际上是更具挑战性的列：什么数和 A 相加结果为 F？振作些；即使是我，在休息日时还必须要想一想呢。

当然，如果上面的数的某一列比底下的数的对应列的值小，问题就出现了，我们除了借位之外别无选择。

借位是小学里死记硬背的学习过程之一，很少有人真正理解它。(理解它，就是不言而喻地接纳令人犯愁和恐怖的新数学。)从某一高度来看，在借位中发生的事情是：从某一列中减少 1，并将之应用到其右侧的列上。我称其为"应用"而不是"加到"是因为在将其从一列移动到它右侧的列时，那个单数被与 10 相乘，这里 10 代表数字基数。(记住，在八进制中，10 代表 8，而在十六进制中，10 则代表 16。)

借位操作本身并没有听起来那么糟糕。看看一个实际的借位操作，你就会明白了：

```
  9 2H
- 4 FH
------
```

这里，最右侧的减法不能完成，因为 F 比 2 大，所以我们需要从左侧的列借一位。

差不多快 50 年了，我仍然记得老修女 Marie Bernard 咬牙切齿地把它写在黑板上，尽管是以 10 为基数："勾掉 9；把它变成 8。然后把 2 变成 12。12 减去 F 是多少，同学们？"是 3，修女。这就是借位的工作原理(我希望可怜的仁慈的修女能够原谅我，因为我把十六进制字节放入了从她口中传出的讲解里)。

从功能上思考一下这里发生了什么。我们从 9 上减去 1，然后把 10H 加到 2 上。一个很容易犯的错误是从 9 上减去 1，然后把 1 加到 2 上，这根本不行(需要我解释吗？)。请这样来思考它：我们正在把某一列的剩余值移给它右侧的列，那里需要一些额外的值。上面的数的整个值不变，(这就是我们为什么将之称为"借"而不是"偷"的原因)，但是贷款的接收者被增加了 10，而不是 1。

借位之后，我们将看到如下内容：

```
 8¹2H
-4 FH
```

(在老修女 Marie Bernard 的黑板上，我们划掉 9，把它变成 8。我刚刚也把它变成了 8。硅芯片比粉笔更有优势，除了 8 的早期生命是 9 这点不那么明显之外。)

当然，一旦到了这个程度，分列减法就都做好了，我们发现二者之差为 43H。

有时有些人可能会问你是否曾经遇到过必须借大于 1 的数的情况。很显然，答案是没有。例如，如果借 2，就应该往接收列上加 20，20 减去任意的单位数，其结果仍然是一个两位数。即所得的差放到一个单列上不合适。减法和加法包含着一个很重要的对称性：

两个数相减，在任意的单列减法中，最多需要借 1 位。

2.6.3 跨多列借位

要理解，你的大部分"旅程"中会遇到很多关于借位的操作，但是，生活就是习惯的过程，你会经常遇到类似这样的减法：

```
  F 0 0 0H
- 3 B 6 CH
```

第一列需要借位，但是第二列和第三列都没有位可借。让我们再次回到小学。修女 Marie Bernard 会像机关枪一样喋喋不休地说："勾掉 F，把它变成 E。把挨着它的 0 变成 10。然后再把 10 勾掉，把它变成 F。把下一个 0 变成 10，明白吗？"(我明白了。在天主教学校里，不明白它的结果简直恐怖得难以想象。)

这里发生的事情是：中间的两个 0 是最右侧的 0 和 F 的贷款代理人，它们以获取足够的值来允许它们本身列能够进行减法的形式来获取"佣金"。每一个最后一列右侧的列都从它的左侧邻居列里借 10，然后再贷 1 给它的右侧邻居列。当所有的借位流过上面的数字时，我们看到这样的结果(减去修女所有的"划掉")：

```
  E F F¹ 0H
- 3 B 6 CH
```

此时，每一列减法都可以进行了，所得的差为 B494H。

在回忆小学“诡计”时，不要陷入古老的十进制规矩，“勾掉 10，把它变成 0。”在十六进制的世界里，10H - 1 = F。所以，应该是勾掉 10，把它变成 F。

2.6.4 意义何在

如果你有一个十六进制计算器，或者支持十六进制显示器的计算器呢？意义在于练习。十六进制是汇编语言的通用语言。对十六进制的理解越深入，学习汇编语言就越容易。而且，如果你凭直觉就能理解十六进制的话，理解机器本身的内部结构也会更加容易。这里，我们正在为以后的学习打下坚实的基础。磨刀不误砍柴工，现在认真学习，后面部分就会省心多了。

2.7 二进制

要想和最奇怪的数制系统——二进制进行较量，十六进制是最好的练习。二进制的基数是 2。通过目前已经学过的数字基数，你都能推测出基数 2 哪些内容呢？

- 每一列的权值都等于它右侧列权值的两倍。
- 在这种基数里，只有两个数字(0 或者 1)。

用二进制计数有点怪怪的，这一点你可以想象的到。就像这样：0、1、10、11、100、101、110、111、1000 等。因为它听起来很荒谬 ，“零、一、一零、一一、一零零”等，所以简单地读出这些单独的数字，然后再在后面加上二进制几个字，会更合理些。例如：在读二进制数 1011101 时，大多数人会读“二进制一零一一一零一”而不是“二进制一百万，十一千，一百零一”——它听起来很大，但实际上对应的十进制值仅为 93。

尽管二进制看起来也许挺奇怪，但是它遵循前面章节中讨论过的所有数字基数的规则。在二进制和十进制之间进行转换的方法和本章前面部分讲解的十六进制和十进制的转换方法完全相同。

在二进制中进行计数接近于对列进行计数(因为只有两个数字)，所以仔细看看表 2-7 会更好一些，这张表展示了二进制数列的权值，一共到 32 位。

表 2-7　值为 2 的幂数的二进制列

二进制数	2 的幂数	十进制数
1	$=2^{0}=$	1
10	$=2^{1}=$	2
100	$=2^{2}=$	4
1000	$=2^{3}=$	8
10000	$=2^{4}=$	16
100000	$=2^{5}=$	32
1000000	$=2^{6}=$	64
10000000	$=2^{7}=$	128
100000000	$=2^{8}=$	256
1000000000	$=2^{9}=$	512
10000000000	$=2^{10}=$	1024
100000000000	$=2^{11}=$	2048
1000000000000	$=2^{12}=$	4096
10000000000000	$=2^{13}=$	8192
100000000000000	$=2^{14}=$	16384
1000000000000000	$=2^{15}=$	32768
10000000000000000	$=2^{16}=$	65536
100000000000000000	$=2^{17}=$	131072
1000000000000000000	$=2^{18}=$	262144
10000000000000000000	$=2^{19}=$	524288
100000000000000000000	$=2^{20}=$	1048576
1000000000000000000000	$=2^{21}=$	2097152
10000000000000000000000	$=2^{22}=$	4194304
100000000000000000000000	$=2^{23}=$	8388608
1000000000000000000000000	$=2^{24}=$	16777216
10000000000000000000000000	$=2^{25}=$	33554432
100000000000000000000000000	$=2^{26}=$	67108864
1000000000000000000000000000	$=2^{27}=$	134217728
10000000000000000000000000000	$=2^{28}=$	268435456

续表

二进制数	2 的幂数	十进制数
100000000000000000000000000000	$=2^{29}=$	536870912
1000000000000000000000000000000	$=2^{30}=$	1073741824
10000000000000000000000000000000	$=2^{31}=$	2147483648
100000000000000000000000000000000	$=2^{32}=$	4294967296

从那个由 0 组成的庄严的金字塔中，我们一眼就能看出这样的内容：根本无望把这个较大的数位想成数字字符串：“10000000……”等。这里急需一个简单表示法，所以，一会儿我会给你提供一个——它的特性会让你大吃一惊。

你也许并不认同，表的最后一行这么大的数不可能在一般的编程中遇到。对不起，但是任意一台像奔腾(甚至它的老祖先 386 和 486)的 32 位微机都可以在一个电脉冲中一口吞下那样的数字，而且吃掉数十亿个那样的数字作为“午餐”。你必须习惯按照类似于 2^{32} 这样的数字进行思考，毕竟，它只是四十亿十进制数字中微不足道的一个。想一下自己的台式电脑的硬盘容量。中等的新款台式 PC 机一般情况下配备的硬盘是 500GB 或更大容量。一个 G 是十亿字节，所以巨大的 32 位数字在你的电脑硬盘上几乎多得不计其数！这个小问题实际上已经刺痛了老牌软件厂商(不对，对不起，应该说是“后遗症”)。20 年前，500GB 容量的硬盘与其说是科幻小说还不如说是空想。而现在，只需花 99.95 美元就能购买那个“空想”。当必须面对一个超过 2GB 可用空间的硬盘时，不止一个 DOS 下的或早期的 Windows 时代的文件工具都绝望地举手投降了。

像八进制和十六进制一样，在使用二进制时，也有标识问题。二进制数 101 与十六进制数 101 不同，与十进制 101 也不同。出于这个原因，二进制数后经常附加一个“B”，以确保阅读程序的人(包括你本人)六个星期后仍然知道正在处理的数制基数是多少。

2.7.1　二进制值

把一个二进制值转换成十进制的方法和把十六进制值转换成十进制的方法相同——实际上，要更加简单，因为你无须思考某一列包含了多少次该列的权值。在十六进制下，你必须要知道权值为 16 的列中包含了多少个 16，等等。在二进制中，某一列的值要么存在(1 次)，要么不存在(0 次)。

通过一个简单的例子会使这一点更清晰些。二进制数 11011010B 是一个在相

对简单的计算机工作中的相对典型的二进制值。(实际上，它比较小——很多常用的二进制数是它的二倍甚至更多。) 把 11011010B 转换成十进制数，涉及借助于表 2-7 对它从右到左扫描，然后记录下任意包含值 1 的列的权值，同时忽略掉那些包含值 0 的列。

清空计算器，开始转换。

(1) 第 0 列包含一个 0；跳过它。
(2) 第 1 列包含一个 1。意味着这个二进制数包含它的权值 2，所以把 2 敲入计算器。
(3) 第 2 列是 0。跳过它。
(4) 第 3 列包含 1。这列的权值为 2^3，或者 8；把 8 加到记录上。
(5) 第 4 列也包含 1；2^4 是 16，把它加到记录上。
(6) 第 5 列是 0。跳过它。
(7) 第 6 列包含 1；2^6 是 64，所以把 64 加到记录上。
(8) 第 7 列也包含 1；第 7 列的权值是 2^7，或者 128。把 128 加到记录上，我们得到了哪个数？218。那就是二进制数 11011010B 代表的十进制值。就这么简单。

从十进制转换到二进制，虽然更加困难些，但是其实现方法和从十进制转换到十六进制的方法一样。返回去，再阅读一下那一节的内容，找到使用的一般方法。换句话说，看看如何做，然后从十六进制参考资料中分离出本质规则。

我敢打赌，你肯定能毫不费劲地把它计算出来。

也许你已经注意到，我是从 0 而不是从 1 开始对这些列进行计数的。计算机领域的一个古怪之处就是经常从 0 开始计数。实际上，称其为古怪是不公平的。计算机的方法是非常有道理的，因为 0 是一个很好的数字，不应该受到歧视。之所以有“裂缝”是因为在我们人类世界中，对某些事物计数告诉我们有多少个该事物，而在计算机世界中，计数通常就是为了命名它们。即我们需要处理第 0 位，然后第 1 位，等等，不只是需要知道到底有多少位。

顺便说一句，这并不是吹毛求疵。当涉及内存地址时，这种情况会一次又一次地发生，像我以前曾经说过的，并且今后仍然要说的一样，这是理解汇编语言的关键。

在编程界中，计数经常是从 0 开始的！

与这一原则发生冲突的一个真实例子能够产生以下问题：我们的千禧年从哪一年开始？大多数人会凭直觉回答 2000 年。但是，严格来讲，20 世纪保持着它沉稳的步伐，一直到 2001 年 1 月 1 日。为什么？因为没有第 0 年。当历史学家

从公元前到公元计算年份时，他们直接从公元前 1 年到公元 1 年，因此，第 1 世纪从公元 1 年开始，结束于公元 100 年。第二世纪从公元 101 年开始，结束于公元 200 年。按照这个顺序扩展，你会发现，20 世纪从 1901 年开始，结束于 2000。相反地，如果我们已经明智地采用当今计算机的方式来计算年份，从公元第 0 年开始，那么 20 世纪应该结束于 1999 年。

把二进制数转换成十进制数，然后将其再转换回来，以此来获得一些实践经验，这是一个很好的想法。通过完成以下转换来锻炼一下你的基本功吧：

```
110
10001
11111
11
101
1100010111010010
11000
1011
```

完成之后，请将下面这些十进制数转换成二进制：

```
77
42
106
255
18
6309
121
58
18,446
```

2.7.2 为什么使用二进制

如果使用整整 8 个数字(11011010)来表示一个平常的三位数，如 218，二进制作为一种基数数制系统，看起来是一项错误的智力投资。当然，对我们来说它会是一个大脑带宽的浪费，即使是只有两个手指的外星人，也很可能有更好的数制系统。

问题是，灯要么处于打开状态，要么出于关闭状态。

这只是另外一种说明计算机是电设备的方式(我将在第 3 章中进行详细讲解)。在电设备中，要么有电压，要么无电压；电流要么流动，要么不流动。在早期，计算机科学家决定，计算机电路中存在电压表示二进制位 1，而电路中缺

少电压则表示二进制位 0。数字虽然不多，但是对于二进制系统来讲足够了。这是我们使用二进制的唯一原因，然而它是一个相当具有强迫性的数制系统，我们被它缠住，无法摆脱。但是，你不必淹没在 1 和 0 的海洋中，因为我已经教给了你一种简写形式。

2.8 二进制简写方式：十六进制

十进制数 218 用二进制表示是 11011010B。然而，用十六进制表示，同样的值就显得很紧凑：DAH。这两个组成 DAH 的 16 进制数字值得我们仔细研究。AH(或像汇编编译器要求的那样：0AH，至于原因，将在稍后给出)代表十进制 10。把任意数转换成二进制只需简单找出它里面包含的 2 的幂数。十进制 10 以内的最大的 2 的幂数是 8。写下 1，同时将 8 从 10 中减掉。剩下 2。现在，4 是一个 2 的幂数，但是 2 里面不包含 4，所以我们在 1 的右侧写下一个 0。下一个较小的 2 的幂数是 2，正好 2 里面包含一个 2，所以，在 0 的右侧再写下一个 1。2-2=0，所以这个数里没有 1 剩下。在最后一个权值为 1 的列上写下最后一个 0。得到以下结果：

```
1 0 1 0
```

回过头去，看看与十进制数 218 等值的二进制数 11011010。最后 4 位是 1010，与 0AH 值相等。

对于 DAH 的上半部分，方法类似。如果采用刚才的方法计算与 0DH 相等的二进制数(它会是一个很好的大脑训练)，结果应该是 1101。按照如下方式观察一下与 218 相等的二进制数：

```
218          十进制
1101 1010    二进制
D    A       十六进制
```

可以把每四位(从右侧开始，不要从左侧开始！)二进制数转换成一个十六进制数，进而实现把长长的二进制 0 1 串转换成一个更为紧凑的十六进制数，这样也许会给你带来“曙光”。

举个例子。这里有一个位数不算超级少的 32 位二进制数：

```
11110000000000000001111101001101110
```

记忆或者操作这样的数位集合是一件非常令人讨厌的事情，所以，让我们将它从右侧开始每 4 位分成一组：

```
1111 0000 0000 0000 1111 1010 0110 1110
```

每一组 4 位二进制数都可以用一个简单的十六进制数来表示。现在开始转换。应该得到如下内容：

```
1111  0000  0000  0000  1111  1010  0110  1110
 F     0     0     0     F     A     6     E
```

换句话说，与那个二进制等值的十六进制数为：

```
F000FA6E
```

当然，我们在使用时应该在后面附加一个 H，同时在前面加上一个 0，所以在任意类型的汇编语言中，这个数实际上应该被写成 0F000FA6EH。

这个数字仍然相当大，但是除了正在做一些诸如统计硬盘空间或其他高位值之类的事情之外，当你还处于汇编语言编程学徒水平的时候，一般情况下这样的 32 位数就是可能遇到的最大量了。

突然间，情况开始看起来有点容易理解了。

十六进制是程序员对于计算机二进制数的简写方式。

这就是为什么我先前说计算机以一种精神分裂的方式同时使用基数 2(二进制)和基数 16(十六进制)。我没有说的是，计算机并不是真的精神分裂，你才是。在计算机内心深处(像我将在第 3 章中讲解的一样)，它只使用二进制。十六进制是一种使你和我更容易处理计算机的方式。幸运的是，每四位二进制数都可以由一个十六进制数表示，所以，对应关系更加清晰和容易理解。

准备计算

到此刻为止，所有的内容都是必要的基础性工作。我已经从概念上解释了计算机是做什么的，并且已经提供了一些工具来理解它们使用的那些有点“外星化”的数字；但是到目前为止，关于计算机实际上是什么，我什么也没有说。在本书中，我们会反复温习十六进制系统，到目前为止，我对于十六进制乘法和位操作(bit-banging)还只字未提。原因很简单：在能够传递一个位之前，必须要知道这个位在什么地方。所以，让我们摘下面具，看看是否能“抓住”一些正在工作中的位。

第 3 章

摘下面具

探索计算机的真正面目

3.1 RAXie，我们怎么不认识你

1970 年 1 月，我正在读高中三年级，芝加哥公立学校已经在校内安装了电脑。一辆满载着昂贵的 IBM 打字机配件的卡车被运送到 Lane Tech，一名困惑的数学老师被抽调到一所充满了喧闹男生的高中讲授计算机科学课(他们有胆量这样叫)。

我很快就领会了它的使用方法。在打卡机上连续猛击一叠愚笨的计算机卡片，将它们放入一个打字机工具——卡片输入箱中，然后怀着敬畏的心情观察打字机在绿色的纸条上跳动它的小高尔夫球，打印出不可避免的错误信息列表。这很有趣。我得的都是齐刷刷的 A。我甚至还保留着我做的第一个能够做点有用事情的程序：一小叠卡片，它能够为手工计算望远镜的反射镜产生抛物线校正因子表，那时我对天文学有着浓厚的兴趣。(我还留着那叠卡片，尽管橡皮圈分解留下的一些黏糊糊的脏东西对读卡器不好，假使那东西依然存在的话。)

一直困扰我的问题是它实际上到底是哪种类型的 RAX(计算机的非常合适的名字)。我们当时使用的是拉姆充电打字机，RAX 控制着这根电话线——我的理解就这么多——但是，RAX 本身是什么呢？

我问了我的指导老师。对话简单整理大至如下：

我：“嗯，老师, RAX 到底是什么？”

他：“啊？嗯，一台计算机，一台电子计算机。”

我：“那是课程笔记上说的，但是我想知道 RAX 是由什么制成的，还有，它是如何工作的？”

他：“嗯，我确信 RAX 是使用电晶体的。”

我：“你的意思是，里面没有杠杆和齿轮？”

他：“嗯，也许有一些，但是里面没有电子管。”

我：“我并不关心电子管。我想它是不是在某处有个计算器；但是，是什么东西能够让它记住 A 在 B 的前面？它是如何知道格式化的含义的？它是如何告诉我们时间的？要想回电话，它必须做哪些事情？”

他：“现在，来吧，那就是为什么计算机如此伟大的原因！他们把它组装在一起，以便于我们不必担心那类问题！谁关心 RAX 是什么？RAX 知道 FORTRAN 并且能够执行任何正确的 FORTRAN 程序，那才是重要的，不是吗？”

他开始流汗，我也是。谈话结束。

那年 6 月，我毕业了。在我的书包里装满了三英寸已调试好且能够工作的 FORTRAN 打孔卡，但是对于 RAX 到底是什么，我仍然毫无线索。

从那天开始，这个问题就一直困扰着我。

Gus 的救赎

6 年后，我正坐在驶往工作场地的 Devon 大街公交车上思考 RAX，膝上放着最新的《大众电子》杂志。封面故事描述了一个名为 COSMAC ELF 的 DIY 项目，它由一个穿孔板(上面布满电路集成芯片，都是通过电线连接)、一些双向开关和一对 LED 数字显示器组成。

它是一台电脑。(嗨，标签上说得真对。)这篇文章的全部内容就是描述如何把它们组装在一起。这些芯片是做什么的？所有的东西都是做什么的？根本不存在什么自动打字机，这几乎令我发疯。

像以往一样，我的朋友 Gus Flassig 在 Ashland 大街上了公交车，坐到我旁边。我问他 COSMAC ELF 是做什么的。他是第一个把一台实实在在的计算机的概念呈现在我面前的人：

这些是内存芯片。你通过将这些双向开关扳到不同的二进制代码状态，把数字加载到内存芯片，扳上来代表一个 1 位，扳下去代表一个 0 位。对 CPU 而言，内存中的每一个数字都代表着某些内容。一个数字让它执行加法；另外一个数字让它执行减法；再另外一个数字让它把不同的数字写入内存，还有很多其他的事情。程序由内存中的一批这种指令数字行组成。计算机读入第一个数，完成这个数告诉它所做的任务，然后读入第二个数，完成这个数告诉它所做的任务，等等，一直到没有数字可用。

如果你没有完全把它搞清楚，不要着急。我具有所有电子爱好者的优点(所以我知道某些芯片的功能)并且已经用 RAX 的 FORTRAN 编写过一些程序。对于我来讲，上帝呀，所有东西一下子击中了要害，并且在我的脑子里爆炸开来，直到气流开始涌出我的耳朵，我知道了！

无论 RAX 到底是什么，我知道，它必须是类似 COSMAC ELF 的东西，只是更大一些而已。我制作了一台 ELF。它更具有教育意义，并且使我能够在更深的层次上理解计算机的原理。我不再建议任何人用散装的芯片一步一步地自己动手组装计算机，当然，疯子除外，尽管那在 20 世纪 70 年代中期相当普遍。

顺便说一句，有些人已经编写了一个基于 Windows 的 COSMAC ELF 模拟软件，它看起来就像我做的那样，并且实际上能够接收和执行 COSMAC 程序。它非常有趣，而且有助于人们了解 1976 年早期的计算机业余爱好。URL 网址如下：

http://incolor.inetnebr.com/bill_r/computer_simulators.htm

网站的作者 Bill Richman 已经重新打印了《*大众电子*》的这篇文章，我就是根据它创建的那台设备。其中的资料不仅引人入胜，而且对于计算原理的芯片级别概念的理解有着非常深刻的教育意义，当时如此，现在依然。

在这一章里，我将与大家分享一些我自己在编写汇编程序过程中的顿悟。(你想知道“硬件”中的“硬”是从哪里来的吗？我敢保证，肯定不是来自于把它猛的一下放到桌子上所发出的声音。)

3.2 开关、晶体管和存储器

开关记忆。

这样来思考它：你把门边墙上的开关打开，天花板中间的灯变亮了。它一直停留在亮的状态。当离开房间时，你又把开关关掉，灯灭了。它持续停留在灭的状态。开关将一直保持当你最后离开它时所处的位置，即使灯泡烧穿了，直到你或者其他人回来，把它拨到另一个位置。你可以通过观看开关把手的位置搞清楚灯是打开的还是关闭的。

从某种意义上讲，开关记住了它的最后一次命令，直到你用一个新的命令“重写”那个命令来改变它。从这种意义上来讲，一个照明开关代表一种类型的基础存储器元素。

照明开关比起电来机械多了。但这并不妨碍它们作为存储器来使用；实际

上，最早的计算机(19 世纪的巴贝奇差分机)就是完全机械的。实际上，他发明的更大但从未完成的版本是蒸汽动力的。巴贝奇的机器有很多小凸轮，这些凸轮可以被其他凸轮从一个位置掷向另一个位置。编码后的数字通过凸轮位置模式来进行记忆。

3.2.1 如果走陆路，就是 1

开关具体是机械的，或电子的，或液压的，或其他什么的，都没有关系。重要的是，一个开关保持着一种模式：打开或者关闭；上或者下；流动或者不流动。我们可以对那个模式赋予一个含义。Paul Revere 告诉他的同伴在旧北方教堂里建立一个代码："走陆路 1，走海路 2"。一旦被点亮，塔尖上的灯就保持亮的状态(这样就记住了那个非常重要的代码)足够长的时间，以便 Paul 去召集民兵，大胜英国人。

因此，一般来说，我们称之为存储器的东西实际上是一个开关集合，它们能够保持一个足够长的状态以便能被人或者机械装置读取和理解。为了我们的目的，这些开关将是电开关，但是请记住，人们已经提议和建造机械计算机和液压计算机并取得了不同程度的成功。

存储器由一些具有可变模式的容器组成，它能保持某一输入模式直到有人或物改变这个模式。

3.2.2 晶体管开关

对于创建基于电灯开关的计算机存储器系统，其中一个问题是，电灯开关非常专业化，它们需要手指去设置，而且它们的输出是电流通路。理想境界是，计算机存储器开关应该被控制它的同样力量去操纵。这就使得存放在某一存储器中的模式能够传递到其他存储器的存储位置。在庞大的电动机械世界里，这样的开关被称为*继电器*。

继电器是一个由电操纵的、用来控制电流的机械开关。我们通过向它输入一个电流脉冲来"掷"继电器，这个脉冲能够为一个小锤提供动力，使其将一个杠杆击打到一边或者另外一边。而这个杠杆能够打开或者关闭一系列电触点，就像普通的电灯开关一样。由继电器制造的计算机已经存在，也许你已经想到了这一点，但那是很久以前的事情，而且(一个典型的继电器的大小与加入饮料用的方形小冰块差不多。)它们并不是很强大的计算机。

真正的电子计算机是用晶体管开关做的。(更早期的计算机也用电子管制造。)晶体管是很小的硅晶体，它利用硅独特的电子属性来充当开关。我不想解释那些独特的电子属性是什么，因为那样它会占用整本书的篇幅。让我们把晶体管开关看作一种类型的用电驱动的黑盒子，并用输入和输出这两个术语来描述它。

图 3-1 展示了一个晶体管开关。(它是一个场效应晶体管，实际上它只是一种类型的晶体管，但却是我们当前计算机所使用的那种类型。) 当电压被加到管脚 1 时，电流在管脚 2 和管脚 3 之间流动。当电压被从管脚 1 中移除时，电流停止在管脚 2 和管脚 3 之间流动。

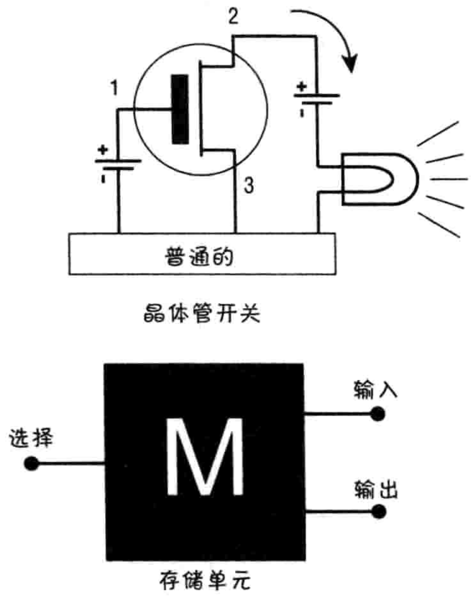


图 3-1 晶体管开关和存储单元

在现实生活中，少数其他部件(最典型的是二极管和电容)也是必要的，它们可以使计算机存储器工作得更平稳。这些是由电线连接到晶体管外边的非必要的小发明(尽管在早期的晶体管计算机中它们是必不可少的)，但是现在被从切割晶体管本身的硅晶体中切割下来，并且几乎不占空间。晶体管开关及其支持部件合在一起被称为存储单元。我已经把存储单元电路的复杂性隐藏在一个合适的黑盒子符号里，如图 3-1 所示。

存储单元将流入它的电流保持一个最小值，因为电流流动会产生热，而热是

电子器件的天敌。存储单元的电路需要精心排列，以便于当向它的输入管脚输入一个微小的电压，同时向它的选择管脚输入一个类似的电压时，该电压会出现并保持在它的输出管脚上。输出电压一直保持它的设置状态，直到将电压从这个单元上整体移除，或者只移除输入管脚的电压而保留选择管脚的电压。

应用到这些管脚的“打开”电压保持一个一致的水平(当然，除非它被整个移除)。换句话说，不要向输入管脚输入 12 伏特电压，然后再将其改成 6 伏特或者 17 伏特。计算机设计者选出一个电压，并且一直坚持使用该电压。此模式本质上就是二进制：你要么向输入管脚施加电压，要么把电压整个移除。输出管脚对此做出回应：它要么保持一个大小为固定值的电压，要么根本没有电压。

让我们为事件状态赋予一个编码：*有电压代表二进制 1，无电压代表二进制 0*。这个编码是随意的。我们也可以说，无电压代表二进制 1，有电压代表二进制 0(并且出于某些说不清楚的原因，某些计算机已经按照这种方式制造)，但是选择权在我们手里。让某些东西的存在状态代表 1 更加自然些，并且它也是主流计算机所采用的方式。

一个单个的存储单元(例如我们这里所说的晶体管存储单元)存放着一个二进制位，或者是一个 1，或者是一个 0。这叫一个位(bit)。位是信息的原子个体。没有半个位，也没有一个半位。

位是一个简单的二进制数，它要么是 1，要么是 0。

3.2.3 难以置信的位缩水

一个位不能告诉我们什么。要想使其有用，我们需要集合很多存储单元。晶体管一开始相当小(最初 20 世纪 50 年代的晶体管大小像小锡兵头顶的大礼帽)，而且一路下来变得越来越小。第一个晶体管由小的锗或者硅芯片做成，大小约九分之一平方英寸。从那以后，晶体芯片的大小没有很大程度的改变，但是晶体管本身已经几乎不可思议地收缩了。

开始时一个芯片只能存放一个晶体管，但半导体设计者们适时地把芯片十字交叉，使之成为 4 个大小相等的区域，并且把每一块做成一个单独的晶体管。从那里添加一些把晶体管转换为计算机存储单元的小器件非常容易。硅芯片非常小而且易碎，通常被放在一个长方形的塑模外壳里，就像一块长着用于电连接的金属腿的小口香糖。

我们现在使用的是一种用电驱动的蛋盒状存储单元：4 个小文件架，每一个可以存放一个简单的二进制位。 然后“收缩”工作开始了。一开始是 8 位，接着

是 16 位，然后是 8 位和 16 位的数倍，都存放在同样的小硅芯片上。到了 20 世纪 60 年代末期，一个硅芯片已经可以存放 256 个存储单元，而且通常以一个 32 行，每行 8 个单元的阵列形式。我的 1976 年的 COSMAC ELF 计算机包含两个存储器芯片。每个芯片上是一个宽为 4 个存储单元，长为 256 个存储单元的阵列。(用图片显示出来是一个名副其实的长卵型塑料盒。)这样，每个芯片可以存放 1024 个位了。

这是那个时代很典型的存储器芯片容量。我们称之为 1K RAM 芯片，因为它能够存放差不多 1000 个比特(bit)的随机存储单元(RAM)。K 来自 *kilobit(千比特)*——即 1000 个比特。稍后我们将重返随机存储器的概念。

20 世纪 70 年代中期，这一伟大的存储器收缩行动进入了更高境界。1K 字节的芯片被分成包含 4096 比特存储空间的 4KB 芯片。这个 4KB 芯片几乎立刻就被分成了 16KB 芯片(16 384 比特的存储空间)。当 IBM PC 机于 1981 年第一次出现时，这些 16KB 芯片成了标准。到了 1982 年，芯片又一次被分割，16KB 变成了 64KB，在那个同样的小口香糖条里，有 65 536 比特的存储容量。记住，我们正在谈论的是把超过 65 000 个晶体管(还有其他零散器件)放到一个每面只有大约四分之一英寸的方形硅片上。

到了 1985 年，64KB 芯片已经被其后继者——256KB 芯片(262 144 比特)取而代之。芯片的容量几乎总在以 4 的幂数倍速度增加，因为当前阶段的芯片被分成 4 个相等的区域，每一个区域所能存放的晶体管数量等同于其上一代硅芯片所能存放的所有晶体管数量之和。

截止到 1990 年，256KB 几乎已经成为历史，1 兆(兆是百万的希腊语)比特的芯片代表了那时的技术发展水平。到了 1992 年，4 兆比特的芯片成为潮流。这个小家伙里面总共有 4,194,304 比特的存储空间，但是，个头仍然不比那条“肉桂色口香糖”大。大概从那时开始，芯片本身变得小而易碎，所以人们通常将 4 个或 8 个芯片焊接到小小的印刷电路板上，以便它们能够在人类笨手笨脚的操作下依然能够安然无恙。

这个游戏还没有结束，目前你能够买到 2G 字节大小——超过 16 亿比特的即插即用电路板存储器模块。

游戏会到此为止吗？不太可能。存储器的容量越多越好，我们正在带来一些难以想象的强大技术，用于创造日益密集的存储器系统。一些物理学家警告说，物理定律也许很快要叫停游戏了，因为现在的晶体管体积如此之小，以至于一次把不止一个电子挤过它们变得越来越困难。那时，一些名为量子力学的真正丑陋

的生活限制开始成为障碍。我们将会找到绕过这些限制的方法(我们总是这么做)，但是在那个过程中，整个计算机存储器的本质也许就改变了。

3.2.4 随机访问

新手有时会发现“随机”是一个关于存储器的令人困惑和烦恼的词，因为随机通常暗含着混乱和不可预测性。这个词的真正意思是“以随机的方式”，表示能够到达一个随机访问的存储器芯片，取出里面包含的任意比特的内容，而不必影响其他内容，就像你能够在公共图书馆中成千上万个书架上随机选择一本书，而不必顺序筛选或打乱书架上的其他书一样。

存储器不总是按照这种方式工作。在采用硅芯片存放存储器之前，它们被存放在某种电磁机器上，通常是旋转磁鼓或者是与我们现在使用的硬盘相差甚远的磁盘。旋转磁鼓存储器发出一个圆形的、位于磁性传感器下方的比特集合。传感器下方每次通过一个比特，如果错过了想要的那个比特，就像一月的芝加哥公交车一样，你只能等它再次到来。这些被称为顺序访问设备。它们以固定的顺序连续地向你展示比特，一次展示一个，你必须按照它的顺序来等待你需要的那个比特。

没必要记它；我们很久以前就不用顺序访问设备做计算机的主要存储设备了。但是我们仍然使用这类系统实现大容量存贮，我会在稍后对其进行讲解。硬盘在本质上属于顺序访问设备。

随机访问是这样工作的：在芯片内部，每一个比特都存放在它自己的(如图 3-1 所示)的存储单元中。每一个此类存储单元都有一个独一无二的编号。这个编号就是一个单元的(因此也是一个比特的)地址(address)。它非常像大街上的某个地址：拐角处的比特是硅巷 0 号，隔壁的比特是硅巷 1 号，等等。你不必敲比特 0 的门，询问哪个比特住在这里，然后再去隔壁问，直到找到想要的那个比特。如果有地址，你可以快速去往那条大街，并在打算访问的那个比特门前停车。

每个芯片都有很多从它引出的管脚，称为地址管脚(address pin)。每个管脚叫作一个数据管脚(data pin)，参见图 3-2。这些地址管脚是传递二进制地址码的电引线。地址是一个只用 0 和 1 表示的二进制数字。我们通过把(比方说)5 伏特编码成二进制 1，0 伏特编码成二进制 0 来将地址应用到地址管脚上。很多计算机硬件也曾使用其他伏特数的电压，而且现在依然在使用。

真正重要的是，我们都同意用管脚上的某一特定电压代表一个二进制位 1。RAM 芯片内部的专用电路将地址解码成芯片内部的大量存储单元中的一个选定

的输入。对于任意应用到这个地址管脚的给定地址而言，只有一个选定的输入会被提高到 5 伏特，进而实现对那个内存单元的选择。

根据打算读取一个比特还是写入一个比特，数据管脚会在存储单元的输入或者输出之间进行切换，如图 3-2 所示。

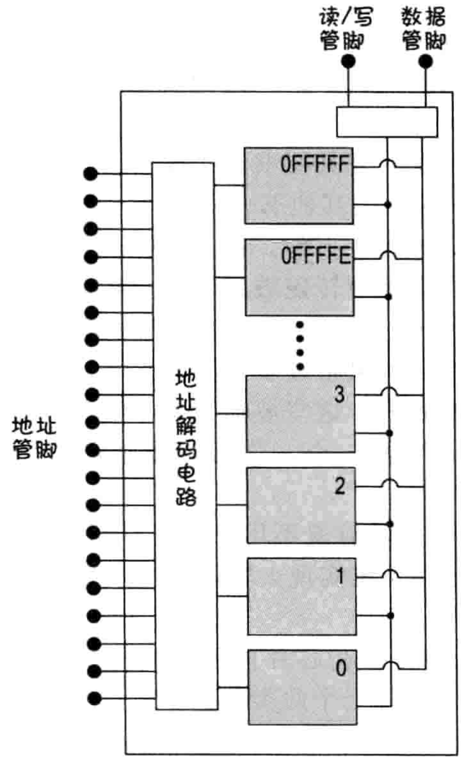


图 3-2 RAM 芯片

这些都是在芯片内部完成的。而在芯片外面呈现的效果是：一旦一个内存地址被应用到地址管脚上，数据管脚就会马上发生变化，瞧！数据管脚将会包含一个电压，代表你需要的位(bit)的值。如果那个位包含一个二进制位 1，数据管脚将会包含一个 5 伏特的信号；否则，将用 0 伏特表示二进制位 0。

3.2.5 存储器访问时间

根据数据在地址应用到地址管脚之后出现在数据管脚上所需的时间，我们对

芯片进行分类。很明显，越快越好，但是有些芯片(又是由于一些很难解释的电子学原因)比其他芯片速度要更快些。

访问时间的值非常小，以至于看起来微不足道：30 纳秒是典型的存储器芯片访问时间。1 纳秒是一秒的十亿分之一，所以，30 纳秒显然比一千万分之一秒小得多。了不起——但是要想完成任何有用的事情，计算机需要访问成百上千的、数百万的或(大多数情况下)数以十亿计的存储器。这些纳秒积少成多。如果已经跻身于汇编语言编程高手的行列，你可能会跳过“金箍”，尽量节省程序执行所需的存储器访问时间，因为存储器访问是计算机性能的一个首要限制因素。实际上，汇编语言专家 Michael Abrash 已经出版了很多关于具体做这件事情的书籍，大多数是在高速图形编程领域。他的建议要点可以(很大程度上)简单地用几个字总结：*只要能不介入内存，就不介入内存*！(你很快会发现这有多难。)

3.2.6 字节，字，双字，四字

一台标配计算机只能拥有一个存储器芯片的时代已经过去很久(实际上，过去几十年了)了。我的可怜的 1976 COSMAC ELF 至少需要两个。当今的计算机需要好多存储器，尽管现在的内存条能存放十亿甚至更多个比特，而不是像 ELF 那样只能存放可怜的 2048 比特。当你想编写高效的汇编程序时，理解计算机如何把它的内存芯片集合在一起形成一个连贯的存储器系统非常重要。有好多种方法可以将内存芯片挂接在一起，但我在这里描述的系统基于英特尔的 PC 兼容机，它已经从 1982 年开始就主宰了台式机的世界。

存储器系统必须能够存储信息。如何将众多芯片组织成一个存储器系统很大程度上由我们组织信息的方式来决定。

答案从*字节*(byte)开始。所有计算机杂志的祖师爷们都争相把这个词作为标题，这一事实表明，它在计算机体系中有着举足轻重的作用。(哎呀，Byte 杂志早在 1998 年年末就停止出版了。)从功能的角度来看，存储器是以字节来度量的。一个字节是 8 位(bit)。两个字节合在一起，构成一个字(word)，两个字合在一起，构成一个双字(double word)。而 4 字(quad word)，你可以想象得到，由两个双字构成，共 4 个字，或 8 个字节。另外，有些人把一组 4 位数据称为比字节还要小点的半字节(nybble)。(这个术语现在用得很少，而且越来越少了。)

这里是一个快速一览表：

- 一位是一个简单的二进制数，0 或者 1。
- 一个字节由 8 位组成。

- 一个字由两个字节组成。
- 一个双字由两个字组成。
- 一个四字是由两个双字组成。

计算机被设计为人类存储和加工信息的工具。人类论文的基本元素由一套符号构成，它包括字母表里的字母(每个都有两种形式，大写和小写)、数字、符号(包括逗号、冒号、句号和惊叹号)等。再加上各种国际字母变异(如：¨a 和`o)，以及很多神秘的算术符号，你会发现，人类信息需要一个大小超过 200 的符号集。(附录 B 提供了各类计算机中使用的符号集。)

字节是体系的核心，这是因为符号集里的每一个符号都可以用一个字节清楚地表示。一个字节是 8 位，2^8 是 256。这意味着一个 8 位二进制数可以表示从 0 到 255 之间的 256 个不同值之一。因为经常需要使用这些符号，所以我们在计算机程序中做的大多数事情使用字节大小的数据块。实际上，除了非常古怪的和特殊类型(比如，我们正在建造的智能食品加工机)的计算机之外，没有计算机处理比一个字节还小的信息。事实上，当今大多数计算机一次处理双字(4 字节，或 32 位)的信息。自从 2003 年以来，PC 兼容机已经能够一次处理 4 字(64 位)的信息了。

3.2.7 精致的芯片排成一行

一件令初学者倍感困惑的事情是：1 个单个的 RAM 芯片甚至存放不了一个字节，尽管它可能包含 5 亿比特。今天我们使用的大部分单独的 RAM 芯片只有 4 个数据管脚，有些甚至只有 1 个数据管脚。整个存储器系统通过把单独的存储器芯片以一种非常巧妙方式组合在一起来创建。

这一点可以通过一个简单的例子窥豹一斑。考虑一下图 3-3。我已经画出了一个存储器系统，它把一个存储的字节分布在 8 个单独的 RAM 芯片上。每一个黑色矩形代表一个如图 3-2 所示的 RAM 芯片。这 8 个芯片每一个存放字节中的一位，而且采用同一个地址。8 个芯片的 20 根地址管脚被连在一起，电学家们也许称之为“并联”。当计算机把一个存储器地址应用到这 20 根地址线时，地址同时出现在存储器系统的所有 8 个存储芯片的地址管脚上。通过这种方式，一个单个的地址可以同时应用到所有 8 个芯片的地址管脚上，在八位数据总线上同时传递 8 个比特的数据，每一个来自于其中一个芯片。

在现实世界中，这样简单的存储器系统已经不复存在，现代的存储器系统有很多种不同的芯片(和它们存储的位) 分布方式。实际上，大多数存储器芯片在每

一个地址存放不止一个比特的数据。每个地址存放 1 个、 2 个、3 个、4 个和 8 个比特数据的芯片已经相对普遍。如何设计快速高效的计算机系统是电子工程领域的一个分支，而且随着存储器芯片不断发展为包含越来越多的存储单元，设计最佳物理存储器系统的方法已经发生了变化。

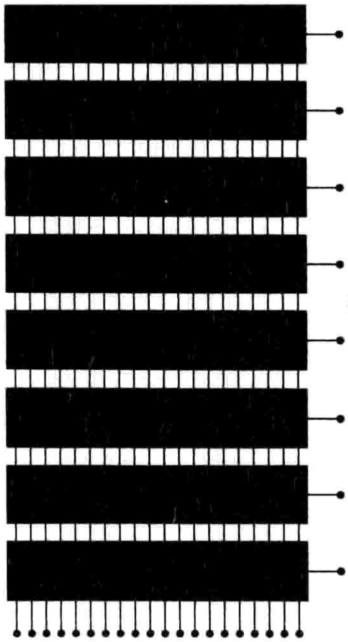


图 3-3 1MB 存储器系统

必须将单独的存储器芯片插到计算机中的时代距离现在已经很久远了。当今的存储器芯片几乎总是聚集在一起，插到各种容量的即插即用双列直插式存储模块(Dual Inline Memory Modules，简写为 DIMM)上。这些模块是小小的绿色电路板，大约 5 英寸长，1 英寸高。在 2009 年，所有的桌面 PC 兼容机都使用这样的模块，而且一般情况下成对使用。每个模块一般在每个地址存储 32 位(通常这样，但也不总是这样，在每个存储器地址处，这 8 个独立的存储器芯片每一个芯片存储 4 位)，两个模块共同工作，在每个存储地址存储 64 位。每个模块内的存储位置数量不同，但是容量通常为 512MB，或者 1GB 到 2GB。(从现在开始，我将使用缩写 MB 和 GB。)

有一点非常重要，需要注意，即将存储器芯片组织成存储器系统的方式并不

会影响程序的操作方式。当已经编好的程序需要访问某一特定内存地址处的一个字节时，计算机会处理将其从芯片丛林的合适位置中取出的过程。以某种方式排列的内存系统从存储器中取出数据的速度可能会比按另一种方式排列的内存系统更快一些，但是地址是一样的，数据也是一样的。从程序的角度来看，没有功能上的差别。

总结：从电学角度来看，计算机存储器包含一行或多行存储器芯片，每个芯片包含很多个由晶体管和其他小电子器件组成的存储单元。多数情况下，为了避免混淆，忘掉这些晶体管，甚至忘掉一行行的物理芯片会很管用。(如此看来，很多中学计算机教师并没有全错，而是误打误撞地由于错误的原因而做对了。)

多年以来，存储器系统一直以不同的方式进行访问。8 位计算机(现在是古董了，而且几乎已经绝迹了)一次访问 8 个比特(一个字节)的内存。16 位计算机一次访问 16 个比特(一个字)的内存。今天的主流 32 位计算机一次访问 32 个比特(一个双字)的内存。基于新的 64 位处理器的高档计算机一次访问 64 个比特(一个四字)的内存。这可能有点让人难以理解，所以在很多情况下，我们可以预想一个具有很长字节大小的容器，每个容器有它自己的唯一地址，这样会好一些。不要假设在一台一次处理一个字的信息的计算机里，只有字才拥有地址。在 PC 体系结构里，惯例是每个字节都拥有它自己的数字地址，不管在一次操作里取出多少个字节。

计算机中每个字节的存储器都有它唯一的地址，即使是在一次处理 2、4 或者 8 个字节信息的计算机中。

如果这看起来似乎不太直观，也许另一个隐喻会对你有所帮助：你去图书馆，取出托尔金(Tolkien)的大部头魔幻作品《指环王》三部曲，这三卷分别有它自己的目录编号(本质上是这个卷在图书馆中的地址)，但是你一下子把它们都拿了下来，并且把它们当成一个实体来处理。你也可以一次只查找其中一本书(如果真想那么做的话)，但是，这样做会要求你额外再去两次图书馆，取回剩余的两卷，毫无疑问，这是对时间和劳动的极大浪费。

32 位和 64 位计算机也是如此。每个字节都有它自己的地址，但是当一台 32 位计算机访问一个字节时，它实际上一次读取从这个字节开始的 4 个字节。你可以使用剩余的 3 个字节，或者如果不需要，干脆忽略它们——如果后来你又决定的确需要使用剩余的 3 个字节，你将不得不再次访问内存以获取它们。为了节省时间，最好一次将它们都取出来。

3.2.8 车间工长和流水线

所有这些关于从存储器中读取数据和写入数据的内容都小心翼翼地避开了这个话题：谁在做读取和写入工作。这个“谁”通常情况下是一个简单的芯片，并且它也是一个非常不同寻常的芯片：中央处理单元，或 CPU。如果你是自己计算机的总裁或者 CEO，那么 CPU 就是你的车间工长，它看着你的命令在各个芯片(工作的具体完成场所)之中得到执行。

也许有些人会说，CPU 是实际完成工作的部件，但是在很大程度上，这种观点过于单纯化了。很多实际的工作是在存储器系统和我们称之为外设(peripheral)的设备中完成的，如显卡、USB 和网络端口等等。所以，尽管 CPU 的确做了很多工作，但它也打包了很多工作给计算机内部的其他部件来做，以便最大限度地让自己做很多更快的、更擅长的事情。像很多优秀的经理一样，车间工长将它能分配的所有事情授权给其他的计算机子系统。

在大部分组装的 PC 中，使用的大多数 CPU 芯片都是由 Intel 公司设计的，20 世纪 70 年代早期的单芯片 CPU 方式几乎都是这家公司发明的。从那之后，Intel 芯片迅速进化，具体内容我将在本章稍后的部分进行讲解。这些年来，在细节上已经有很多改变，但是从某一高度来看，任何 Intel 或 Intel 兼容的 CPU 所做的事都大致相同。

3.2.9 对话内存

CPU 芯片最重要的工作就是与计算机存储器系统进行通信。像存储器芯片一样，CPU 芯片也是一个方形硅片，上面有很多晶体管——时至今日，已达数亿个！这个易碎的硅芯片被放在一个塑料或者陶瓷的盒子里，并从里边伸出很多电子管脚。像存储器管脚一样，CPU 的管脚传递用电压值编码的信息，这个电压值典型情况下为 3 到 5 伏特。管脚上有 5 伏特电压代表二进制位 1，管脚上有 0 伏特电压代表二进制位 0。

像很多存储器芯片一样，CPU 芯片有很多专门用于内存地址的管脚，这些管脚被连接到计算机系统的存储器芯片上。我已经将这些内容画在了图 3-4 中，并且 CPU 芯片左侧的存储器系统与图 3-3 中所示的相同。当 CPU 需要从内存中读入一个字节(一个字、双字、四字)时，它把该字节的内存地址放到它的地址管脚上，编码成一个二进制数。几纳秒之后，所需的字节就出现(也是以二进制的形

式)在存储器芯片的数据管脚上。CPU 芯片也有数据管脚，它通过自己的数据管脚“啜食”存储器芯片提交的字节。

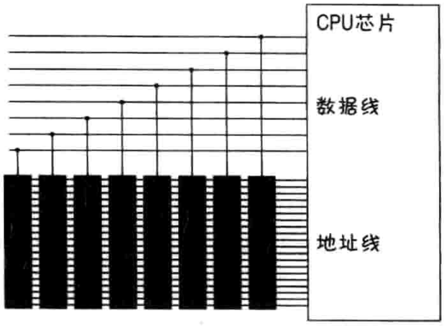


图 3-4　CPU 和存储器

当然，这个过程也可以反向进行：为了向内存中写入一个字节，CPU 首先把它想写入的存储器地址放到它的地址管脚上。几纳秒以后(根据系统速度及涉及的内存数量，不同的系统所需的时间会有所不同)，CPU 把它想写入的字节放到自己数据管脚上。存储器系统乖乖地把这个字节存放到请求的地址中。

当然，图 3-4 是纯概念层面的。现代的存储器系统比这里展示的复杂得多，但是在本质上，它们都采用同样的工作方式：CPU 把一个地址传到存储器系统，然后存储器系统或者接收来自 CPU 的数据，并将其存储到那个地址，或者把那个地址中找到的数据放到计算机的数据总线上，供 CPU 处理。

3.2.10　驾驭数据总线

CPU 和存储器系统之间的这种给-拿(give-and-take)活动代表了计算机内部发生的大部分事情。信息从存储器中流到 CPU 中，再从 CPU 中流回来。信息也可以按照其他路径流动。计算机还包含一些额外设备，叫作外设(peripheral)，它们或者是信息源或者信息的目的地(或者二者皆是)。

显卡、硬盘、USB 和网络端口是个人电脑中最常用的外设。像 CPU 和存储器一样，它们都是靠电驱动的设备。大多数现代外设包含一到两个大芯片，也许包含一对儿支持大芯片的小芯片。像 CPU 和存储器芯片一样，这些外设都包含地址管脚和数据管脚。一些外设(尤其是图形卡)还拥有自己的存储器芯片，目前它们使用专用的 CPU。(现代的高性能图形卡是一个拥有自主权的高性能计算机，尽管它只拥有一个特定而有限的使命。)

外设能够与 CPU“谈话”(即它们传递给 CPU 数据或把数据从 CPU 中取出)，有时也互相通信。这些对话发生在链接地址管脚和数据管脚的电连接上，所有计算机中的设备都有通用的管脚。这些电线被称为数据总线(data bus)，它们形成了一种类型的合用线，把 CPU 连接到计算机的所有其他部件上。一个复杂的电仲裁系统决定这些不同的设备能够在什么时间以什么顺序使用这根合用线来互相“交谈”，但是它经常以同样的方式发生：地址被放到总线上，后面跟着一些数据。(一次多少数据取决于所涉及的外设。)总线上发出一些特定的信号，用以指明该地址是代表存储器中的一个位置，还是一个绑定到地址线的外设。为了和前面一直讨论的存储器地址区分开来，我们将外设地址称为 I/O 地址(I/O address)。

在大多数个人计算机中，数据总线是出现在扩展槽(extension slot)中的主要元素，而很多外设(尤其是图形卡)就是这些插入扩展槽中的印刷电路板。通过数据总线在扩展槽中的电管脚实现的连接，外设可以与 CPU 和内存进行对话。

尽管扩展槽使用起来很方便，但是，它们也在计算机内部产生了一些延迟。简单来说，外设只是计算机内部的主要电路板(主板)上的某个角落处的几个芯片。

3.2.11　车间工长的口袋

每个 CPU 都包含一些存储数据的地方，称为“寄存器”(register)。这些寄存器是车间工长的口袋兼工作台。当 CPU 需要一个地方来临时存储某些东西时，它就可以使用一个空的寄存器。CPU 也经常把数据存到内存中，但要比存到寄存器中花费更多的时间。因为寄存器实际上位于 CPU 的内部，把数据放入寄存器或者再将其读出来都很迅速。

更重要的是，寄存器是车间工长的工作台。当 CPU 需要将两个数相加时，最简单也最方便的方法就是把数据放入两个寄存器，然后将这两个寄存器相加。用二者之和(通常也在 CPU 里)代替两个相加的原始数据之一，但之后数据还可被放入另一个寄存器，或者仍然与另一个寄存器中的另一个数相加，或者存出到内存中，或者参与很多其他操作。

CPU 的即时工作过程在名为寄存器的临时存储容器中进行。

涉及寄存器的工作通常速度很快，因为寄存器位于 CPU 的内部，并且被专门连接到其他寄存器及 CPU 的内部装置上。几乎不太需要数据移动，而且移动的距离也不必太远。

像存储单元一样，实际上，像整个 CPU 一样，寄存器是由晶体管制造的，

但是寄存器拥有很多独立的名字，例如 EAX 或者 EDI，而不是有很多数字地址。使情况变得更加复杂的是，尽管所有的 CPU 寄存器都有很多特定的共同属性，但有些寄存器拥有着特殊的、其他寄存器所不具备的权利。理解 CPU 寄存器的行为和局限性与遵循中东和平过程有点相似：有一些合作、联盟，通常还有一些每个寄存器遵循的让人眼花缭乱的机密“议程表”。不存在描述这类内容的一般系统，像西班牙语中的不规则动词一样，你只能下功夫记住它们。

大多数外设也有寄存器，外设寄存器的使用范围比 CPU 寄存器的使用范围更小一些。它们的“议程表”也更加明确，没有充满智慧的秘密。但是这并不意味着它们理解起来更容易，这一点每一个尝试过在寄存器级别上为图形卡编程的人都可以证明。幸运的是，近年来，所有与外设进行的通信都交由操作系统处理，详情参见第 4 章。

3.2.12 流水线

如果 CPU 是车间工长，那么外设就是流水线上的工人，数据总线就是流水线本身。(然而，与大多数流水线不同，车间工长在流水线上比他的手下工作得更卖力！)

假设信息通过网络端口外设进入计算机，网络端口从计算机网络电缆上接收数据位并将其“装配”成代表字符和数字的数据字节。然后网络端口将装配好的字节放到数据总线上，CPU 从它上面获取该信息，将其记录下来或者以其他方式进行处理，然后再把它放回数据总线。之后，显卡接收来自数据总线的字节，并将其写到显存中，以便我们可以在显示器上看到它。

这是一个朴实无华的简单描述，但是很明显，在这个“盒子”里面进行着很多事情。而沿着数据总线和 CPU 以及外设之间的、连续不断的高速通信就是完成计算机工作的幕后英雄。问题随之而来：是谁告诉车间工长及其手下员工该做什么事情的？是你。你怎样做到的呢？你编写一个程序。程序从哪里来呢？它在存储器中，与很多存放在存储器中的数据一起。实际上，程序也是数据，它是我们所知的整个编程思想的核心。

3.3 遵循计划行事的盒子

最后，我们来到计算的本质，即：程序的本质及其如何指示 CPU 控制计算机完成交办的工作。

我们已经了解了内存如何存储信息字节。这些字节都是二进制编码，它们是以极小的电压值存储的位 1 和位 0 模式，这些模式共同组成二进制数。我们也谈到过符号，以及某些特定二进制编码如何被解释为对于我们人类有意义的某些内容，例如字母、数字、标点符号等等。正如字母表和数字代表对我们人类有意义的某些内容一样，代码集对 CPU 也意味着某些内容。这些代码被称为机器指令(machine instruction)，并且它们的名称可通过其实际内容(即下达给 CPU 的指令)来进行记忆。当 CPU 正在执行一个程序时，它从数据总线上取出一系列数字，每次取出一个。每个数字告诉 CPU 做某件事情。CPU 知道如何做这件事情。当执行完某条指令时，它取出下一条指令并执行。它继续执行直到某些东西(程序中的一条指令，或诸如复位按钮的之类的电信号)告诉它停下来。

让我们看一两个对于所有来自 Intel 的现代 IA-32 CPU 芯片都通用的例子。8 位二进制编码 01000000(40H)对于 CPU 来说，意味着一条命令：*向寄存器 AX 中的值加 1，并且将和存回 AX*。大多数机器指令不止占用一个字节。很多指令占用两个字节，更多指令占用 4 个字节。二进制代码 11010110 01110011 (0B6H 073H)构成另外一条指令：把数值 73H 加载到寄存器 DH 中。而二进制编码 11110011 10100100 (0F3H 0A4H)则能够引导 CPU 完成以下事情(深呼吸一下)：开始把由 CX 寄存器指定数量的字节从存放在寄存器 DS 和 SI 中的 32 位地址中取出来，并存入存放在 SI 和 DI 中的 32 位地址处，每移动一个字节后，同时更新 SI 和 DI 中的地址，并将 CX 中的值减一，最后当 CX 中的值变为 0 时，停止执行。

现在不必记这些特定指令的所有细节，在本章后半部分我们将重返机器指令。Intel IA-32 CPU 所能理解的其余几百条指令的复杂性和功能大致在这两者之间。有些指令能够完成算术运算(加法、减法、乘法和除法)和逻辑运算(与、或、异或等)，也有一些指令能够将信息移入内存。有些指令能够“操纵”被执行的程序逻辑的执行路线。有些指令拥有一些鲜为人知的功能，并且很少在操作系统内部之外的其他地方出现。现在需要记住的要点是：*每一条指令一般情况下告诉计算机执行一个较小且有限的任务*。很多条指令被按顺序提交给 CPU，以引导其完成更为复杂的任务。而写出那些指令序列实际上就是汇编语言编程。

让我们进行更深一步的探讨。

3.3.1 取指和执行

计算机程序无非是存放在内存中的一张机器指令表。这张表没有什么特别的地方，它在内存中的存储位置也没有什么特别的地方。它可能存在于任何地方，

而且表中的数据也不过是一些二进制数而已。

组成计算机程序的二进制数只是在 CPU 处理它们的方式上有些特殊之处。当一台现代的 32 位计算机开始运行时，它从商定的内存地址中取出一个双字。(现在，这个开始地址是如何商定的并不重要。)这个每行包含 4 个字节的双字被从内存中读出并且加载到 CPU 中。CPU 检查这个双字中包含的二进制位模式，然后开始执行这条取出的机器指令命令其所执行的任务。

古老的基于 8088 的 8 位机器(比如原始的 IBM PC)一次只取出一条指令，而不是 32 位的奔腾级别机器的 4 个字节。由于大多数机器指令尺寸比一个单个字节大，在开始真正遵循指令并执行指令交给的任务之前，8088 CPU 不得不返回存储器去取第二个(或第三个、第四个)字节来完成这条机器指令。

CPU 一执行完一条指令，就转到存储器，按顺序取出下一条指令。在 CPU 内部，有一个特殊的寄存器，名为指令指针寄存器(instruction pointer)，它包含下一条即将从内存中取出和执行的指令的地址。每当一条指令执行完毕后，这个指令指针都被更新为指向内存中的下一条指令。(在现代 CPU 内部，正在进行着一些神奇的硅魔术，它能够“猜”出下一条即将执行的指令，并且将其放到架子一侧，所以在取指时，它就会在那里，不过速度更快了——但就结果而言，前面描述的关于它的过程也没有错。)

所有这些都像时钟一样规规矩矩地执行。计算机有一个以电驱动的子系统，叫作系统时钟(system clock)，它实际上是一个能够以非常精确的时间间隔发出方波脉冲的振荡器，CPU 内部的很多极小的晶体管开关根据系统时钟产生的脉冲来协调它们的行为。在过去，经常需要几个时钟周期(一般来说，来自于时钟的脉冲)来执行一条简单的指令。随着计算机速度不断变快，大部分机器指令能够在一个单个时钟周期内完成。现代的 CPU 能够并行执行指令，因此在一个单个的时钟周期内可执行多条指令。

所以，过程是这样进行的：取指和执行；取指和执行。CPU 在指令指针寄存器的指引下在内存中行进。当行进时，它是这样工作的：在内存中移动数据，在寄存器中移动数据，把数据传给外设，用算术运算和逻辑运算处理数据。

计算机程序是存储在内存中的二进制机器指令的列表。在内存中，它们与其他数据字节的列表没有任何区别，只是当被取出时，CPU 对它们的解释有所不同。

3.3.2　车间工长的内脏

我在前面指出，机器指令是二进制代码。我们在前面章节中经常将其一带而

过，但是为了理解 CPU 的真正本质，我们必须摒弃将机器指令当成数字的偏见。它们不是数字。它们是被设计用来掷电开关的二进制模式。

在 CPU 内部，有相当多的晶体管。(我的台式电脑里的 Intel Core 2 Quad 包含 582 百万个晶体管，那些拥有超过十亿个晶体管的 CPU 芯片现在使用还比较有限。)这些晶体管中的很少一部分构成了车间工长的口袋：用于存放信息的计算机寄存器。很多晶体管构成一种叫作高速缓存(cache)的临时储存器，我会稍后对它进行讲解。(目前，请把高速缓存看成一个存储架的小集合，它们总在车间工长的手边，这样车间工长不必穿过房子就可拿到更多的资料。)然而，这些晶体管的更主要部分是一个一个互相连接的开关，在一个令头皮发麻的复杂网络中，被连接到更大数量的开关上。

一条极为简单的机器指令 01000000 (40H)引导 CPU 将存放在 AX 中的数值加 1，然后再将结果存回 AX。当考虑计算机的真实特性时，以这种方式来思考机器指令 01000000 的执行非常有意义。

CPU 从内存中取出一个字节。该字节包含二进制编码 01000000。一旦这个字节已经完全位于 CPU 内部，CPU 就在本质上将这条机器指令字节压入 8 个晶体管寄存器。只有一个孤单的位 1 把它的开关以电的方式“打开”，剩下的所有数字(都是位 0)把它们的开关“关闭”。

根据连锁反应，这八个开关翻转前几十个开关的状态，然后几百个，然后几千个，甚至有些情况下数以万计的 CPU 内部的开关状态。这不是随机的——在 CPU 内部发生的众多纳秒级电行为完全是根据 Intel 工程师小组蚀刻在 CPU 的硅芯片上的模式来操纵的。最后，也许在掷了数千个单个的开关之后，包含在 AX 寄存器中的值突然间比它原来的值大了 1。

至于它是如何发生的，这很难解释，但是必须记住，CPU 内部的任意数字都可被看成是一个二进制编码，包括存放在寄存器中的值。同时，大部分 CPU 内部的开关包含不止一个手柄。这些开关，称为门(gate)，它们根据逻辑规则进行工作。为了让某个开关通过门来实现“关闭”，也许需要两三个或更多个“打开”开关投掷同时到达某个特定的门。

这些门被用于在 CPU 内建造复杂的内部机制。一些门的集合能够在名为加法器(adder)的设备中把两个数相加，而所谓的加法器，无非又是很多个共同工作的小开关，这些小开关构成门，然后很多门共同工作，构成加法器。

作为二进制编码 01000000 投掷的开关队伍中的一部分，AX 寄存器中的值被转储到加法器中，同时数字 1 被输入到加法器的另一端。最后，产生一个开关投

掷波，新求得的和从加法器中出现，而后回到寄存器 AX 中——工作完成。

所以，计算机的车间工长就是由开关组成的——就像计算机的所有其他组成部分一样。它包含着很多数量惊人的这种开关，而这些开关以更加令人吃惊的方式互相连接。不管你感到困惑也好，或者是仅仅被它搞得精疲力竭(像我在休假日时一样)也好，重要的事情是，CPU，最终是电脑，精确地完成我们告诉它去做的事情。我们在内存中以表的形式建立一个机器指令列表，然后，天啊，我们的哑巴硅芯片“朋友”突然活跃起来，开始“挣钱维持生计”。

3.3.3　改变航向

计算机本质的第一个真正魅力是：内存中的一串二进制编码一步一步地告诉计算机做哪些事情。第二个魅力是皇冠上的真正宝石：*有一些机器指令能够改变机器指令被取出和执行的顺序。*

换句话说，一旦 CPU 执行完一条有用的机器指令，下一条机器指令也许会告诉 CPU 回去继续重复执行该条指令，再执行，再再执行，需要多少次就执行多少次。CPU 能够记录某一特定指令或指令列表被执行的次数，一直重复执行它们，直到遇到一个已经预置的计数值。此外，如果这些指令根本不需要再执行，它还能够安排跳出这一特定机器指令序列。

这意味着存储器中的机器指令列表不一定是从上面开始，然后毫无偏差地运行到底部。CPU 能够执行第一个五十或一百或一千条指令，然后一下子跳到程序的结尾——或跳回到起始位置，再次开始执行。它能够非常平稳且迅速地跳出这个列表和在这个列表中弹上弹下。它能执行一些当前位置上面的指令，然后突然转到下面某些其他地方，执行更多的指令，然后再转回来，捡起它离开时剩余的部分，决不错过任何一条指令或者浪费过多的时间。

这是如何完成的呢？回想一下，CPU 包含一个特殊的寄存器，它总是包含下一条即将执行的指令的地址。这个寄存器就是指令指针寄存器，它与任何其他的 CPU 寄存器没有本质的区别。像存在一条机器指令能够把 1 加到寄存器 AX 上一样，也存在另外一条机器指令能够将存放在指令指针寄存器中的地址加上或减去某个数。向指令指针寄存器里加上 100，在继续工作之前，CPU 会从这个机器指令列表的当前位置立即往下跳转 100 个字节。从指令指针寄存器里减去 100，CPU 会从这个机器指令列表的当前位置立即往上跳回 100 个字节。

最后，第三重打击：*CPU 能够以根据已经完成的工作，改变自己的执行路线。* CPU 能够根据存放在内存中的值，或者根据名为标志寄存器(flag)的 CPU

寄存器中的某一单独位的特定状态，决定是否执行某一给定的指令或指令组。CPU 能够计算出它需要做某事多少次，然后按照那个数字指定的次数做这件事。或者它能够做某些事情，然后再开始做它，不断地一次又一次重复做它，每次都要检查(通过看某些地方的某些数据)一下任务是否已经完成，还是不得不再执行一轮。

所以，你不仅能够告诉 CPU 要做什么，还能够告诉它去哪里做这些事情。更棒的是，你有时能够让 CPU 像一名忠实的警犬一样，嗅出以更快的可能方式完成工作的最佳前行路线。

在第 1 章中，我将一个计算机程序描述为一系列步骤和测试。CPU 所能理解的大多数机器指令是步骤，而其余的就是测试了。测试通常是双向测试，而且实际上，关于要做什么的选择通常相同：跳转，还是不跳转。仅此而已。你可以在 CPU 内部进行很多次不同的条件测试，但是，选项通常是跳到程序中的另一位置还是继续原来的路线。

3.4　是什么 vs.怎么做：体系结构和微体系结构

本书实际上主要侧重于 Intel 的 32 位 x86 CPU 及其他公司制造的兼容 Intel 的 32 位 CPU 的汇编语言编程。有很多不同的 Intel 和 Intel 兼容的 x86 CPU 芯片。完整的列表包括 8086、8088、80286、80386、80486、奔腾、奔腾 Pro、奔腾 MMX、奔腾Ⅱ、奔腾 D、奔腾Ⅲ、奔腾 4、奔腾 Xeon、奔腾Ⅱ Xeon、奔腾 Core、赛扬，还有很多其他的 CPU 芯片，很多是专用的、鲜为人知的、短命的。(你曾经听说过 80376 吗？)而且，那些只是 Intel 设计和销售的 CPU。一些其他公司(主要是 AMD)也有自己设计的 Intel 兼容 CPU 芯片，它们能给前述列表再添加几十项。在一个 CPU 型号中，通常还会有另外三四个具有一些醒目名字的变种，例如：核心、卡特迈、酷睿等。更糟糕的是，还可能有奔腾Ⅲ核心和赛扬核心。

谁能记住这么多呢？

答案是没有人能够真正做到。为什么？因为几乎在所有情况下都不需要大量的细节。CPU 的灵魂被清楚地分为两个部分：CPU 做什么事情；CPU 如何做这些事情。程序员从外面来看它：CPU 做什么事情。设计计算机主板和其他包含 Intel 处理器硬件系统的电子工程师和系统设计人员需要知道其余内容，但是他们是极少数非常吃苦耐劳的一群人，他们知道自己是谁。

3.4.1 体系结构的演变

我们程序员从外面看到的东西有：CPU 寄存器、CPU 能够理解的机器指令系列、专用子系统，例如可能包含自己指令和寄存器的快速数学处理器。Intel 对所有这些内容都进行了详细的定义，并且将其在线发布或者编写到书里以便程序员能够学习和理解它们。这些定义统称为 CPU 的体系结构。

随着时间的推移，CPU 的体系结构也在不断地进化，供应商们不断添加新的指令、寄存器和其他一些特性到产品线中。理想情况下，这些进化都是向后兼容的，这意味着新特性一般情况下不会取代、禁用或改变老特性的外在效果。Intel 的主打产品线非常擅长于向后兼容，上至 1978 年的 8086 CPU，下至现在各种具有特定限制的“x86”系列，甚至为古老的 8086 编写的程序在现代的 Pentium Core 2 Quad CPU 上仍然能够运行(不兼容性更多情况下与不同的操作系统相关，而非 CPU 细节本身)。

当然，反之则不正确。近年来，在 Intel 的 x86 产品线上逐渐引进了一些新的机器指令。例如，1996 年首次引入的机器指令无法被 1993 年设计的 CPU 识别；但是，1993 年首次引入的指令几乎总是存在，而且能够在新式的 CPU 上照常运行。

除了向指令集周期性地添加内容之外，体系结构本身也在突飞猛进。这些飞跃的典型情况体现在 CPU 的“宽度”上。1986 年，随着 80386 CPU 的引入，Intel 的 16 位体系结构扩展到了 32 位，并且添加了大量的指令和操作模式，CPU 寄存器的宽度增加到原来的两倍。2003 年，x86 体系结构继续扩展，这次是到 64 位，而且又添加了很多新的指令、操作模式和扩展寄存器。然而，依附于扩展的 64 位体系结构的 CPU 也能够运行为老式的 32 位体系结构而编写的软件。

Intel 的 32 位体系结构叫 IA-32，也是本书要讲解的对象。由于特殊的原因，新的 64 位体系结构叫 x86-64，主要是因为它并非 Intel 原创。2000 年初，Intel 的主要竞争者 AMD 创造了一个向后兼容的 64 位 x86 体系结构，它做得如此之好，以至于 Intel 不得不屈尊采用它(Intel 自己的 64 位体系结构叫 IA-64 Itanium，它不能向后兼容 IA-32，而且被市场全盘否定)。

除了一些小毛病之外，新的 64 位 Intel 体系结构包含 IA-32 体系结构，而依次地，IA-32 体系结构包含更古老的 16 位 x86 体系结构。了解哪个 CPU 添加了哪些指令到体系结构中是很有用的，记住，当使用“新”指令时，你的代码不能运行在早期制造且尚未添加该新指令的 CPU 上。然而，这一问题是可以解决

的。对于程序而言，有一些方法可以知道 CPU 有多新，进而将自己限定在 CPU 目前的属性中。同时，还有一些即使了解也没有任何用处的内容。

3.4.2　地下室里的秘密机制

由于涉及向后兼容问题，除非有非常好的理由，否则 CPU 的设计者们不会再添加新的指令或寄存器到体系结构中。有一些其他的更好的方法来改善 CPU 家族。这些方法最重要的是要提高处理器的吞吐量，它不仅仅是单纯地提高 CPU 时钟频率的问题。另一个是要降低电量消耗。这不仅仅是一个“绿色”的问题。CPU 使用的有些电量是热浪费，而且这些热浪费如果不被最小化的话，能够把 CPU 芯片“烹熟”，并且破坏周围的器件。因此，设计者们经常在寻找一些方法来减少完成同样任务所需的电量。

提高处理器吞吐量意味着提高同一时间段内 CPU 执行的指令数目。很多神秘的技术与提高吞吐量有关，并拥有诸如此类的名字：预读取、L1 高速缓存、L2 高速缓存、分支预测、超管道和微操作熔合等。有些技术能够降低或消灭 CPU 的内部瓶颈，以便使 CPU 和存储器系统能够几乎一直保持忙碌状态。另外一些其他技术则延伸了 CPU 的能力，使之一次能够处理多条指令。

总的来说，CPU 为了完成指令让它做的事情而依赖的所有电机制统称为 CPU 的微体系结构(microarchitecture)。它是你看不到的地下室机制。工厂车间工长的比喻在这里已经不合适了，我们得再采用另外一个比喻。

假设你拥有一家为福特公司制造变速箱组装零件的公司，你有两个独立工厂。 一个已经 40 岁了，一个刚刚建立。两个工厂都精确地建造相同的部件——它们必须这样，因为福特公司把这些产品放到它的变速箱里，根本不知道或者不关心到底是两个工厂中哪一个制造的。这样，在千分之十英寸内，凸轮或外罩是一样的，不管它们是老工厂制造的还是新工厂制造的。

老工厂已经拥有相当长的历史了，但是新工厂是在 40 年来经营旧工厂过程中学到的所有经验的基础之上设计和建造的。它拥有更符合逻辑的布局、更好的照明、更现代的需要、更少的人操作，而且不用调整就能工作更长时间的自动化工具。

结果是：新工厂可以更快、更高效地制造凸轮和外罩，同时浪费更少的电量和原材料，而且需要更少的工人。你可以基于运营第二个工厂过程中学到的所有经验创建一个甚至更加高效的第三个工厂，总有那么一天，你会把第一个工厂关闭。

虽然如此，凸轮和外罩仍然是一样的，不管它们在何处制造。具体它们是如何制造的，福特公司或任何其他人并不关心。只要凸轮在同样的偏差下以同样的尺寸制造就行，至于“如何制造”并不重要。

所有的工具、流水线布局和每个工厂的一般结构都可视为工厂的微体系结构。每建立一个新厂，该新工厂的微体系结构都能更加高效地完成老工厂一直做的事情。

CPU 也是如此。Intel 和 AMD 经常重新设计它们的 CPU 微体系结构，使之变得更加高效。其中的驱动力是不断提高的硅制造技术，这些技术能够把越来越多的晶体管放到一个单个的 CPU 芯片上。更多的晶体管意味着有更多的开关，对于同样的吞吐量和电效率的老问题，有更具潜力的解决方案。

当然，在提高微体系结构方面的最高指示是：不要破坏已经存在的程序，这一点可通过修改机器指令或寄存器的操作方式来实现。这就是为什么它是地下室里的秘密机置的原因。CPU 的设计者们不遗余力地保持着 CPU 做什么和如何在 5000 万个晶体管之间具体完成那些事情之间的界限。

所有醒目的代号，如 Conroe、Katmai、Yonah，实际上表明微体系结构有了些许调整。在微体系结构方面的主要改变也拥有自己的名字：P6、NetBurst、Core 等等。已经有在线文档对其进行了详细描述，但是如果你还无法做到与时俱进，也不必忧伤。我大多数时间也在用手指(搜索相关资料)紧跟其发展。

我所讲的一切都是为了让你(作为一个新的程序员)不受 Intel 微体系结构的更多影响(与你应该受到的影响相比)。在如何编写程序代码方面，微体系结构细节的不同很少(可能，几乎从来不)带来派上用场的优势。微体系结构并不神秘(你可以从网上在线获取很多关于它的资料)，但是为了清醒起见，应该暂时把它看成一个神秘的家伙。现在，我们有很多更重要的东西需要学习。

3.5　工厂经理

到目前为止，我所描述的内容与其说是“计算机”，不如说是“计算”。一台功能性计算机不仅仅是一个能够执行简单程序的 CPU。我在 1976 年建造的 COSMAC ELF 设备是一个实验品，充其量是一种类型的教育玩具。它是一个拥有一些存储器和足够电量支持的 CPU，(通过开关和 LED)我能够向其输入机器码，观察在内存芯片里正在发生什么。它离有用这个词还差得很远。

我的第一台有用的计算机出现在两年之后。它拥有一个键盘、一个 CRT 显

示器(尽管不能处理图形)、两个 8 英寸软驱和一台打印机。它非常有用，我用它撰写了很多杂志文章以及前三本书。我为它编写了很多应用程序，比如最初的 WordStar 字处理软件；但是真正使它变得有用的却是其他一些东西：操作系统。

3.5.1 操作系统：角落办公室

操作系统(operating system)是一个程序，它管理着一个计算机系统的运行。在包含一系列由 CPU 执行的机器指令方面，它像任何其他程序一样。操作系统的不同之处在于它们拥有一般情况下不会交给诸如字处理软件和电子表格之类的程序的特殊权利。如果我们继续使用把 CPU 当成工厂车间工长这个比喻的话，那么操作系统就是工厂经理。整个实实在在的工厂都在它的控制之下。它监督原材料进入工厂；它监督工厂内部工作的进行(包括车间工长所做的工作)，包装已完成的产品并运输给客户。

实际上，我们早期的微机操作系统并不是很强大，也做不了太多的事情。它们旋转磁盘，处理存储在磁盘驱动器中的数据，当需要时，将数据送回磁盘。它们接收来自键盘的按键敲击，然后把字符发送到显示器。另一种较少的情况是，它们能够把字符发送给打印机。这就是操作系统。

1979 年，CP/M 是台式微机的最新操作系统。如果通过键盘输入一个程序的名字，CP/M 就会访问磁盘，把该程序从一个磁盘文件加载到内存中，然后真正把对机器的所有权利移交给这个已经加载的程序。当 WordStar 运行时，它在内存中覆盖操作系统，因为内存非常昂贵，所以数量不是很多。一次只能运行一个程序，一点也不夸张。CP/M 不会返回(到内存中)直到 WordStar 退出。然后 CP/M 被重新从软驱中加载，等待来自键盘的另一条命令。

3.5.2 BIOS：是软件，但并不软

如果当 Wordstar 退出时，CP/M 不在内存中，那么是什么把它带回到内存中的呢？很简单：WordStar 重新启动电脑。实际上，每次一个软件运行时 CP/M 都离开。每次软件退出时，该软件都会重启这台机器，于是 CP/M 又回来了。CP/M 非常小，从软盘重新启动它只需不到两秒的时间。

随着计算机系统变得越来越快，内存变得越来越便宜，操作系统也与字处理软件和电子表格一起，不断得到完善。当 IBM PC 出现时，PC DOS 很快代替了 CP/M。PC 拥有足够的内存，所以，当程序被加载时，DOS 不会离开(内存)，而

是恰恰相反，仍然驻留在内存原地。DOS 能够比 CP/M 做更多的事情，而且也不比它大很多。这是可能的，因为 DOS 能够访问一些 CP/M 不能访问的特定 PC 编码。

IBM 已经将处理键盘、显示器和磁盘驱动器的程序代码单独拿出来并烧制到一种特殊类型的、名为只读存储器(ROM)的存储器芯片上。一般的随机存储器在断电时内容就消失了。而 ROM 不管有没有电都会保持它的内容。这样，成千上万条机器指令就不必从磁盘中加载了，因为它们总是位于一个焊在主板上的 ROM 芯片里。ROM 上的软件称为基本输入输出系统(Basic Input /Output System, 简写为 BIOS)，因为它能够处理计算机的输入设备(例如键盘)和输出设备(例如显示器和打印机) 。

有些人将那些存储在具有非易失性的 ROM 芯片上的(例如：BIOS)软件昵称为固件，因为尽管它仍然是软件，但又不完全是软件，嗯，软件的软是指它存放在内存里或者磁盘上。所有现代计算机都有 BIOS 固件，尽管如今的 BIOS 软件能够做的事情与 1981 年的已经大不相同。

3.5.3 多任务魔术

DOS 的“任期”很长。Windows 的第一个版本并不是一个全新的操作系统，而是一个以图形模式描绘在显示器上的简单文件系统和程序启动装置。在图标的底层，DOS 仍然在那里，做它一直做的事情。

直到 1995 年，Windows 才从根本上发生改变。那一年，微软发布了具有崭新图形用户界面的 Windows 95，但是在底层机制上已经有些内容发生了非常彻底的变化。Windows 95 工作在 32 位保护模式下，至少需要 80386 CPU。(我将在第 4 章详细讲解“保护模式”的含义。)目前，我们可这样理解保护模式：它明确指定操作系统为老板，而不再仅仅是字处理软件和电子表格的同类。Windows 95 没有充分使用保护模式，因为它仍然拥有 DOS，需要处理 DOS 应用程序，并且这样的“遗留问题”软件在保护模式成为一种可选方案之前很久就已经写好了。然而，Windows 95 的确拥有一些以前 PC 世界中从未见过的内容：*抢先式多任务处理(preemptive multitasking)*。

到了 1995 年，内存已经变得相当便宜了，同一时间出现在内存中的不再只是一两个程序，而是多个。在与 CPU 煞费苦心的合作中，Windows 95 营造了一种令人相信的错觉：内存中的所有程序在同一时间内运行。这是通过为每一个加载到内存中的程序赋予一个 CPU 时间片来完成的。程序在 CPU 上开始运行，它

的若干条机器指令将会得到执行。

然而，经过一段时间之后(通常是几分之一秒)，Windows 95 从第一个程序中“抢回”对 CPU 的控制权，然后将其交给列表中的第二个程序。那个程序将会执行几毫秒指令直到它对 CPU 的控制权被操作系统“抢回”。Windows 95 将会沿着这个列表传下去，让每个程序都运行一会儿。当到达列表的结尾时，它又重返列表首部，继续运行这个程序列表，如此循环，让每个程序运行一段时间。CPU 的速度非常快，以至于坐在显示器面前的用户会认为所有的程序都在同时运行。

图 3-5 将使这个过程变得更清晰。想象一个旋转开关，在它里面有个轮子在不停地连续旋转，顺序接触若干个触点中的每一个，每一圈接触一个。每当它接触到一个程序的触点时，该程序就被允许运行。当轮子移到下一个触点时，前面的程序就停在它的磁道上，下一个程序获取一点时间开始运行。

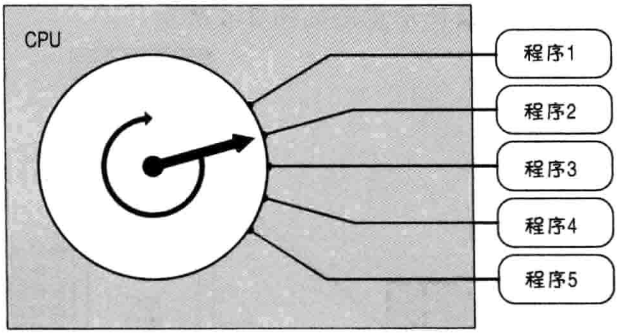


图 3-5　多任务思想

操作系统能够为列表中的每一个程序赋予一个优先级，以便有些程序能够比其他程序获取更多的执行时间。高优先级任务获取更多的时钟周期，而低优先级任务则获取较少的时钟周期。

3.5.4　内核提升

Windows 95 的多任务能力创造了很多应用，但是在 1995 年，很少有人听说过名为 Linux 的类 Unix 操作系统，它最早由一个名叫 Linus Torvalds 的年轻芬兰人编写，并于 1991 年发布。

Linux 没有像 Windows 95 一样精美的图形用户界面，但是它能够处理多任务，而且内部具有更为强大的结构。Linux 的核心是一个名为内核(kernel)的代码

块，它充分利用了 IA-32 保护模式的优点。Linux 内核与图形用户界面是完全分开的，它被保护起来以防被系统某处非正常工作的程序损坏。系统存储器或者被标识为内核空间(kernel space)，或者被标识为用户空间，运行在用户空间的任何内容都不能被写入(一般情况下也不能从内核空间中读取)存放在内核空间的任何内容中。内核空间和用户空间之间的通信通过严格控制的系统调用来进行处理(本书后面会对此做更多的介绍)。

只有运行在内核空间的软件才能直接访问物理硬件(包括内存、显存、外设)。希望使用系统外设的程序只能通过内核模式设备驱动来实现访问。

微软在 1993 年发布了自己的受 Unix 启发的操作系统。Windows NT 的内部结构与 Linux 有很多相似之处，内核和设备驱动运行在内核空间，其他所有程序则运行在用户空间。这种基本设计现在仍然沿用于 Linux 和 Windows NT 的后继者中，例如：Windows 2000、Windows XP、Windows Vista 和 Windows 7。真正的保护模式操作系统的一般设计示意图如图 3-6 所示。

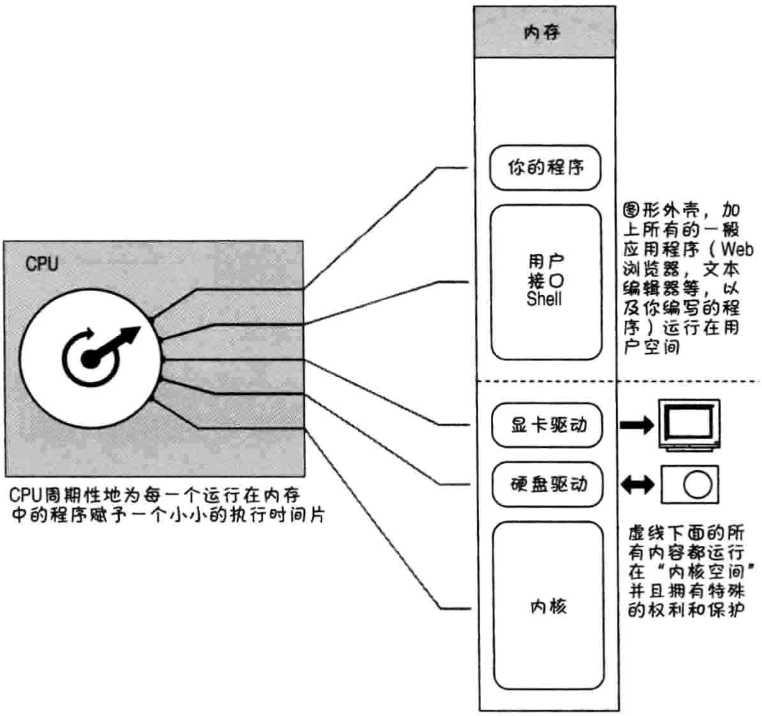


图 3-6 一个成熟的保护模式操作系统

3.5.5 内核爆炸

2000 年初，拥有两个 CPU 插口的台式 PC 开始销售。通过使用“对称多处理(SMP)”机制，Windows 2000/XP/Vista/7 和 Linux 都支持在一个单系统中使用多个 CPU。当所有处理器都一样时，多处理是“对称的”。当可以使用两个 CPU 时，操作系统在大多数情况下在一个 CPU 上运行自己的代码，而在另一个 CPU 上运行用户模式应用程序。

随着技术的进步，Intel 和 AMD 能够把两个相同但完全独立的代码执行单元放到一块芯片上。其结果是产生了第一个双核 CPU：AMD Athlon 64 X2 (2005)和 Intel Core 2 Duo (2006)。到了 2007 年，四核 CPU 开始得到普遍应用。(本书是在 Intel Core 2 Quad 6600 上撰写的。)当然，四核以上的 CPU 是可能的，但是，如何使用这样一笔令人为难的财富仍需要更多的讨论，现在它还是一个颇具争议的话题。

3.5.6 计划

我借用一个前面曾经说过的关于计算机运行机制的最具说服力的比喻对所有内容加以总结：*计算机是一个能够执行计划的盒子*。这是 Ted Nelson 的原话，他是颇具影响力的 *Computer Lib/Dream Machines* 一书的作者，少数拥有令人生气的狂妄自大习惯的人之一。

你编写计划。计算机通过向 CPU 一个字节一个字节地传递指令给来完成它。在底层，这个过程是令人生畏的复杂的电连锁反应，包括成百上千的由很多个成百上千甚至成千上万个晶体管组成的开关。然而，那部分是对你隐藏的，所以不必为之担心。一旦你告诉那堆晶体管做什么，它们自己就知道如何去做。

这个计划(即内存中的机器指令列表)就是汇编语言程序。本书的核心内容就是教给你如何将这些机器指令正确地排列到内存中供 CPU 使用。

如果运气好，到现在为止，你应该已从概念上基本理解了计算机的用途以及计算机的定义。是时候开始近距离观察机器指令引导 CPU 执行的操作的本质了。如同计算中的所有内容一样，操作大部分是在存储器中完成的，这些存储器既包括主板上的常规内存，又包括那些记忆之王——CPU 寄存器。

第4章

位置，位置，位置

寄存器、内存寻址以及了解数据存储位置

之所以写这本书，很大程度上是因为我找不到一本至少让我钦佩的汇编语言入门书。几乎所有关于汇编语言的书都是从介绍指令集的概念开始，然后开始一条一条地讲述机器指令。这是低能的，这类书籍的作者应该受到惩罚。即使你已经学会了指令集中的每一条指令，你依然没有学会汇编语言。

你甚至还没有入门。

对“CPU 存在的目的即为执行机器指令”这一天真的想法很容易驳斥：只有 CPU 的“电子手”中先有了机器指令，它才能执行这些机器指令。CPU 的真正工作以及汇编语言的真正挑战在于在内存中定位这些所需的指令和数据。只有傻子才学习机器指令。(这样的人很多。)*汇编语言的诀窍在于对内存寻址的深刻理解*。其他所有内容都是细节——而且是很简单的细节。

4.1 内存模式的乐趣

内存寻址是一项困难的任务，而 x86 CPU 系列产品中存在很多种不同的内存寻址方式这一事实使其变得更加困难。这些方式中的每一种都叫作内存模式(memory model)。最新的 x86 CPU 系列产品主要使用的内存模式有三种，还有少数这三种基础之上的变种，尤其是中间那种的变种。

在 32 位 Linux 下编程基本上只限于一种内存模式，一旦对内存寻址深入理解之后，你将会非常喜欢它。然而，在这里我打算把所有这三种模式都进行详细讲解，尽管三重奏中的两个老式方法早已进了博物馆。不要跳过对这些博物馆成员的讨论。通过研究化石来了解各种各样的生物如何随时间进化，将有助于更好地理解今天的生物。同样，了解一些关于老式的英特尔内存模式的知识，将有助

于对今天可能使用的内存模式有更加直观的理解。

在本章结尾，我将会简单介绍一下 64 位内存模式，它只是小荷才露尖尖角，目前的使用还不是很广泛。在本书中以及若干年后，32 位保护模式仍将是主流。

最老式的也是现在最古老的内存模式叫实模式平面模型(real mode flat model)。它完全僵化，但是相对简单。年纪较大的(现在已经退休了的)内存模式叫实模式段模型(real mode segmented model)。它可能是你在各种类型的编程中(如：汇编，或者其他语言)学到的最令人讨厌的东西。DOS 编程的最高境界会用到这种实模式段模型，其结果是使程序员们备受折磨。最新的内存模式叫保护模式平面模型(protected mode flat model)，它是很多现代操作系统(如 Windows 2000/XP/Vista/7 和 Linux)背后的内存模型。注意，保护模式平面模型只能用于支持 IA-32 体系结构的 386 及更新的 CPU 中。8086、8088 和 80286 不支持。Windows 9*x* 有点位于两种模式之间，我怀疑除了微软的员工之外，很少有人能够真正理解它内存寻址的所有方式——甚至连微软自己的员工也不能。Windows 9*x* 经常崩溃，在我看来，其中一个主要原因就是它采用了一个十分愚蠢的内存模型。(动态链接库，又名 DLL——人类计算机史上的瘟疫——是另外一个主要原因)。它那怪诞的内存模式虽然不是你不应该考虑在 Win 9x 编写汇编程序的唯一原因，但是却是最好的一个；假设 Windows 9x 现在老老实实地走在成为化石的小路上，你可能永远也不必在它上面编写汇编程序。

我在本书中采用了一个策略，在进入正题之前，我先把它简单陈述一下：我一开始将讲解如何在可用于 DOS 下的实模式平面模型上进行内存寻址。它学习起来出奇地容易。我之所以讲解实模式平面模型，是因为你经常无意中会在这里或者那里碰到它，你需要理解它，即使从来没有为它编写过一行代码。今天真正工作的并且在不久的将来也仍将工作的是 32 位保护模式平面模型，采用这种模型的操作系统有 Windows、Linux 或者任意的实模式 32 位保护模型操作系统。整个工作的关键是：*实模式平面模型非常像保护模式平面模型的缩影。*

有一个大平面模型和一个小平面模型。如果掌握了实模式平面模型，学会保护模式平面模型将不费吹灰之力。一旦掌握了后者，你将成为内存寻址的真正主人。

让我们来看一下这个疯狂的家伙是如何工作的。

4.1.1 16 位将带来 64K 存储空间

1974 年，我刚刚从大学毕业，Intel 推出了 8080 CPU 并初步发明了微计算

机。(是的，我是一个老家伙，但正是因为如此，我才幸运地亲历了这段历史。) 8080 是那时令人痴狂的小东西。我有一个运行速度为 1MHz 的 8080 计算机，它是一个非常高效的字处理器，我使用这台电脑做的主要事情就是字处理。

8080 拥有一个 8 位的 CPU，这意味着它一次能够处理八位的信息。然而，它却拥有 16 根地址线。CPU 的位数——它的通用寄存器应该包含多少位——十分重要，但是在我看来，衡量 CPU 效率的最重要标准是它一次操作能够调集多少根地址线。在 1974 年，16 根地址线已经足具挑战性了，因为那时内存极其昂贵，大多数计算机最多拥有 4KB 到 8KB 的内存(记住，那意味着 4 000 或 8 000 字节)——有些计算机甚至更少。

16 根地址线可以寻址 64K 字节。如果用二进制来计数(计算机经常这样做)的话，你将被限制在 16 个二进制列里，你能够从 0 计数到 65 535。(口语中的“64K”就是 66 536 的简称。)这意味着 65 536 个单独的内存位置中任意一个都拥有唯一一个从 0 到 65 535 之间的地址。

8080 的内存寻址方案非常简单：把一个 16 位地址放到地址线上，然后取出存放在那个位置处的 8 位值。注意：内存系统中的地址线的根数和存放在每个位置上的数据的位数之间没有必要的关系。8080 在每个位置存放 8 位二进制数，但是它也能够在每个位置存放 16 位或者 32 位数据，并且仍然使用 16 根内存地址线。

迄今为止，8080 主要使用的操作系统是 CP/M-80。CP/M-80 有些与众不同，它出现在已安装内存的顶部——有时是为了便于包含在 ROM 中，但更主要的是为了让路，从而允许临时程序拥有一致的内存起点，临时程序(不像操作系统)被加载到内存中，只有需要的时候才运行。当 CP/M-80 从磁盘中读入一个程序并运行时，它将程序加载到距离内存底部 256 个字节的地址为 0100H 的地方。内存的前 256 个字节叫程序段前缀(Program Segment Prefix，简称 PSP)，里面包含着各种各样的零碎消息和一个用于程序磁盘输入输出的通用内存缓冲区。可执行代码本身一开始并不运行，直到系统对 0100H 进行寻址。

我已经把 8080 和 CP/M-80 内存模型画在了图 4-1 中。

8080 的内存模式(比如 CP/M-80 中使用的)非常简单，并且经常被人们使用；所以当英特尔创建它的第一个 16 位 CPU——8086 时，它想让人们更加容易地将较老的 CP/M-80 软件从 8088 翻译到 8086——这个过程叫移植(porting)。要想做这件事情，一种方法是确保 16 位寻址系统(例如 8080 中的寻址系统)仍然能够工作。所以，尽管 8086 能够寻址的范围是 8080 的 16 倍(16×64KB = 1MB)，英特

尔将 8086 创建为可让程序只占用那 1M 内存里的 64K 字节段，并且完全在它里面运行，就好像它是小型的 8080 内存系统一样。

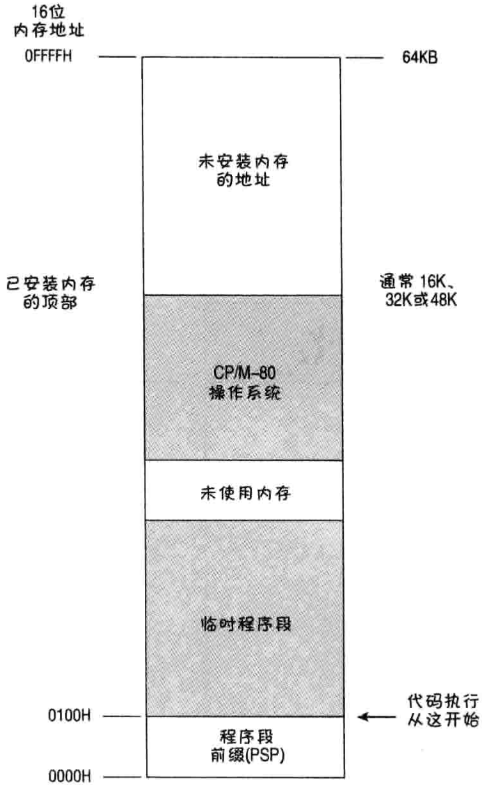


图 4-1　8080 内存模式

这是通过使用段寄存器(segment register)来完成的，从根本上来讲，它们是位于 CPU 寄存器内部的内存指针，指向内存中的事情开始的位置，这里所说的“事情”，可能是数据存储，也可能是代码执行或者其他内容。很快我们将学习更多有关段寄存器的知识。目前，将它们看成指示某处地址的指针就足够了，在 8086 内存的 1M 字节内，一个从 8080 世界移植过来的程序即将开始运行(见图 4-2)。

谈到 8086 和 8088，我们需要考虑 4 个段寄存器(再次的，我们很快将会详细介绍它们)。为了图 4-2 的目标，考虑名为 CS 的寄存器——它代表代码段(code segment)。再次重申，它是指向 8086 的一兆字节内存中的某一位置的指针。这个位置作为 64K 内存区域的起始点，在这个区域中，一个快速转换的 CP/M-80 程

序能够流畅地运行。

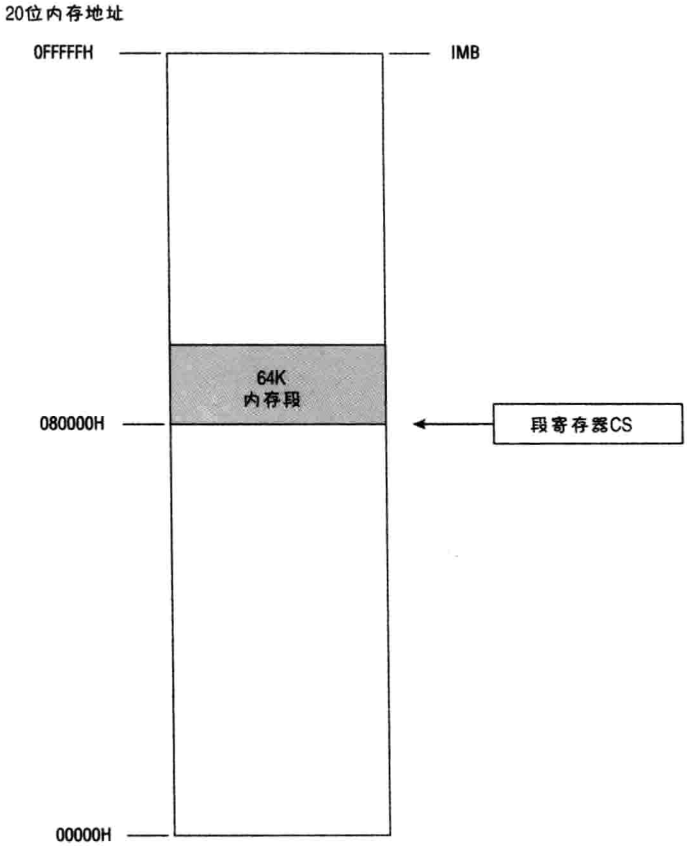


图 4-2 8086 内存系统里的 8080 内存模式

从短期来看，这是一种非常明智的考虑——同时，从长期来看，这也是一种非常糟糕的设想。两年之内，许多 CP/M-80 程序被转换成了 8086 程序。但是，当程序员们企图从头开始创建全新的、与 8080 从未谋面且不需要段模型的程序时，问题变得很严重。糟糕——段模型控制着 8086 的体系结构。一些一次需要大于 64KB 内存的程序不得不使用 64KB 的内存块，通过切换进入和离开段寄存器的值来在实现在这些内存块之间的切换。

这是一场噩梦。没有好的理由要学习它，但是，理解实模式段地址寻址方式有助于理解另外两个 x86 平面模型的工作机制，在这个过程中，你能够逐渐深入地理解 CPU 的本质。

4.1.2 兆字节的本质

当运行在分段实模式下时，x86 CPU 最多能使用 1MB 的可直接寻址内存。这个内存也叫实模式内存(real mode memory)。正如第 3 章中简单描述的那样，1MB 存储器实际上并不是一百万字节，而是 1 048 576 字节。像简写术语“64K”一样，即使是在以十为基数的数制系统中也不会出现 1M 字节，因为计算机操作的是基数 2。这 1 048 576 个字节用以 2 为基数的数制系统来表示是 100000000000000000000B 个字节。那是 2^{20}，稍后我们将会继续讨论这一事实。打印数字 100000000000000000000B 太过庞大，最好采用兼容的(而且更紧凑的)16 基数来表示它，也就是第 2 章中已经讲过的 16 进制数制系统。2^{20} 与 16^5 相等，可以用十六进制数 100000H 来表示(如果对数字基数的概念仍然感到糊涂，说明你还没有掌握它，建议重温第 2 章，——即使已经掌握了也可再次阅读)。

现在，这里有一个棘手且至关重要的问题：在一个包含 100000H 字节的内存“银行”中，最后一字节在这个内存“银行”中的地址是什么？答案不是 100000H。线索对于那个问题而言，起的只是负面作用：什么是内存中的第一个字节的地址？这个答案，你可能还记得，是 0。计算机总是从 0 开始计数。(人类一般从 1 开始计数。)这种脱节一次又一次地发生在计算机编程中。从计算机编程的角度来看，在一个由 4 项组成的行中，最后一项的编号是 3，因为在一个包含 4 项的行中，第一项的编号是 0。编号值分别为：0、1、2、3。

内存中一个字节的地址只是从零开始计数的那个字节的编号。这意味着一个包含 1M 字节的内存中的最后一个，也就是最高的地址是 100000H 减 1，或 0FFFFFH(尽管从数学的角度来看，初始值零不是必要的，但是它仍然在那里，为汇编编译器提供方便，并且有助于防止汇编程序令人迷惑不解。要养成在以 A 到 F 开头的十六进制数的前面添加 0 的习惯)。

那么，1MB 内存地址为从 00000H 到 0FFFFFH。采用二进制表示法，相当于从 00000000000000000000B 到 11111111111111111111B 的范围。这是一个很大的位数，确切地说，20 位。如果参考第 3 章中的图 3-3，你会看到一个具有 20 根地址线的 1M 内存。这 20 位地址就被引导到这 20 根地址线上，因此，任何一个可以表示为 20 位的地址都能且只能标识内存块中包含的 1 048 576 个字节之一。

这就是 1M 字节内存：由包含 20 根电线的地址总线所连接的、计算机内部的一些内存芯片组合。通过向那 20 根地址线提供 20 位地址来标识 1MB 中的一个字节。

4.1.3　向后兼容和虚拟 86 模式

现代的 x86 CPU(例如：奔腾)能够在比这更大的内存范围内进行寻址，稍后我将简要解释一下它如何实现以及为什么要这样实现。对于 8086 和 8088 CPU 而言，计算机真正拥有 20 根地址线和一兆内存。386 及其之后的英特尔 CPU 能够寻址 4 兆内存，而不必把它分成更小的内存段。当 32 位 CPU 工作在保护模式实模型下时，一个段是 4GB——所以大多数情况下，一个内存段足够。

然而，很多使用分段技术编写的 DOS 软件仍然随处可见，我们必须对其加以处理。所以，为了维护与古老的 8086 和 8088 的向后兼容(backword compatibility)，较新的 CPU 被赋予一种权利，使其能够将自己限制在老式 CPU 能够寻址和执行的范围之内。当一个奔腾级别的 CPU 需要运行为实模式段模型编写的软件时，它采用一项简洁的技术，暂时把自己变成一个 8086。这叫虚拟-86 模式(virtual-86 mode)，它对 DOS 软件提供了非常好的向后兼容性。

当在 Windows NT 及更高版本下启动一个 MS-DOS 窗口或“DOS box”时，你就在使用虚拟 86 模式，等同于在 Windows 保护模式存储器系统内创造一个小小的实模式岛屿。它是保持向后兼容的最佳途径，其中的原因你很快就会明白。

4.1.4　16 位眼罩

在实模式段模型里，x86 CPU 可以“看到”整整 1MB 的内存。也就是说，CPU 芯片能够将自己设置为只能使用 32 个地址引脚中的 20 个，进而向存储器系统传递 20 位的地址。从这个角度来看，解决方案似乎非常简单明了。然而，在理解实模式段模型时，你可能遇到的大部分麻烦来自这一事实：虽然这些 CPU 可以看到整整 1M 字节内存，但是它们被限制为只能使用 16 位眼罩来看那 1M 字节。

眼罩这个比喻可能比你想象中的更加接近原义。看一下图 4-3，那个长长的矩形代表 1 兆字节的内存，CPU 可以在实模式段模型下对其进行寻址。CPU 位于右侧。中间是一个隐喻的纸板，在它上面切出一个 1 字节宽，65 536 个字节长的插槽。CPU 可以在其内存系统的长度范围内向上和向下滑动那块纸板。但是，在任意时间里，它都只能访问 65 536 个字节。

CPU 看到的实模式段模型下的内存很特殊。它被限制为以块的方式查看内

存，这些块中的每一个长度都不超过 65 536 字节——再次提醒，也就是我们所说的“64KB”。利用这些块——也就是说，知道目前正在使用哪一个块以及如何从一个块移动到另一个块——是实模式段模型编程所面临的真正挑战。现在是时候进一步研究段的概念以及它们的工作机制了。

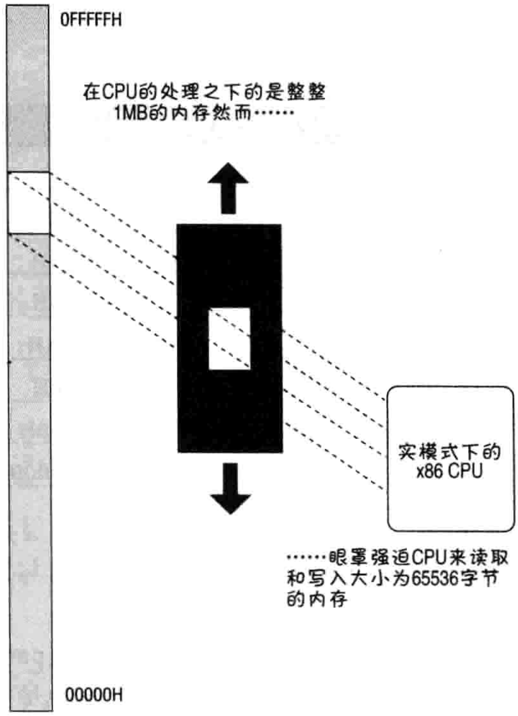


图 4-3　通过 64K 眼罩看 1MB 内存

4.2　段的本质

到目前为止，我们已经非正式地把段(segment)看作 CPU 能够查看和使用的、位于较大内存空间里的、一块一块的内存。在实模式分段模型中，段是一个内存区域，它从一个段落(paragraph)边界开始，并且扩展一定数量的字节。在实模式分段模型中，这个数量小于或者等于 64K(65 536)。你已经见到过数字 64K，但是你见到过段落吗？

暂停一下，我们先上一堂关于古老的 86 家族琐事的课。段落(paragraph)是一

个内存度量，等于 16 个字节大小，它是用于描述不同内存容量的众多技术术语之一。我们以前已经见过了其中一些，它们都是 1 个字节的偶数倍。请记住，字节是原子数据；松散的内存位更像是亚原子粒子，如果包含它们的一个字节(或更多字节)“缺席”，这些内存位就根本不存在。有些术语的使用频率较其他的高，但是你应该知道所有这些术语。表 4-1 提供了所有这些术语。

表 4-1　内存术语集合

名称	十进制值	16 进制值
字节	1	01H
字	2	02H
双字	4	04H
四字	8	08H
十字节	10	0AH
段落	16	10H
页	256	100H
段	65 536	10000H

其中一些术语(例如十字节)很少出现，有一些(例如页)几乎从未出现过。段落(paragraph)这个术语起初很少见，并且大多数情况只与段开始处的内存位置有关。

任何能被 16 整除的内存地址都被称为一个段落边界(paragraph boundary)。第一个段落边界是地址 0。第二个是地址 10H；第三个是地址 20H，以此类推。(记住，10H 等于十进制的 16。)任何一个段落边界都可以被认为是另一个段落的开始。

这并不意味着一个段真正从每 16 个字节开始，向上和向下贯穿整个 1 兆内存。段像是现代可调书柜中的其中一层隔板。在书柜背面有很多间距为 0.5 英寸的插槽。每个隔板可以插入任意插槽。但是，没有成百个隔板，只有四五个。几乎所有的插槽都是空的和未使用的。它们存在的意义是：当需要时，能够让少量的隔板可以沿书柜上下调整高度。

以一种非常类似的方式，段落边界就是插槽，可以以它为起点开始一个段。在实模式段模型中，一个程序可能只使用四五个段，但是这些段中的每一个可能开始于 65 536 个段落边界之中的任何一个，当然，这些段落边界位于这 1MB 在实模式段模型中可用的内存里。

这次又出现了数字 65 536，也就是我们深爱的 64K。有 64K 不同的段落边界，可以从这些段落边界开始一个段。每一个段落边界都有一个编号。这些编号照例从 0 开始，一直到 64K-1，用十进制表示是 65 535，用十六进制表示是 0FFFFH。因为一个段可能开始于任一段落边界，所以把段开始的那个段落边界的编号叫作该段的段地址。

其实，我们很少谈到段落或段落边界。如果看到与实模式段模型相关的术语——段地址，请记住，两个相邻的段地址之间并不是紧挨着，而是相距 16 个字节(即 1 个段落)之远！在图 4-4 中，每个阴影条都是一个段地址，每 16 个字节开始一个段。最高的段地址为 0FFFFH，它距离实模式的 1MB 内存的最顶部 16 个字节。

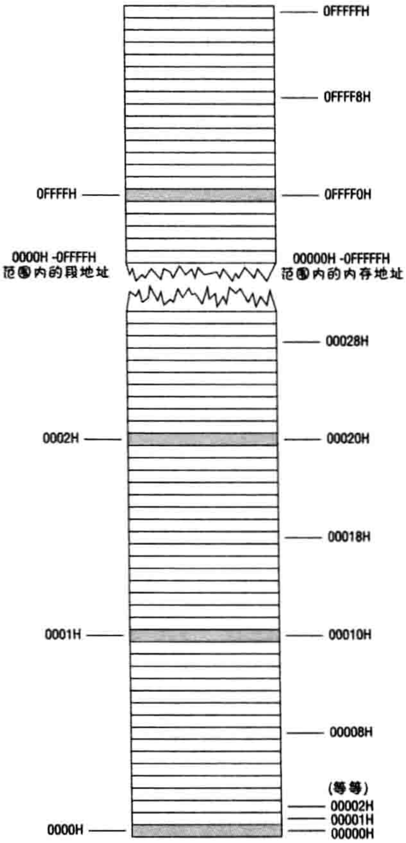


图 4-4 内存地址和段地址

总之，段可以开始于任意段地址。共有 65 536 个段地址均匀地分布在实模式下的满满 1MB 内存中，每两个段之间相隔 16 个字节。与其说段地址是一个强制，不如说它是一个许可；对于所有 64K 个可能的段地址而言，只有五六个在某一时段被真正用作段的开始。可以将段地址想象成可以放置段的插槽。

关于段地址就讲这么多；那么，段本身是什么呢？关于段的理解，最重要的一点是，它最大可达 64K 字节，但不一定非要那么大。一个段可能只有一个字节长，或者 256 个字节长，或者 21 378 个字节长，或者任意小于 64K 的字节长。

4.2.1 一个界限，而非一个位置

我们主要通过声明一个段的开始位置来定义它。那么，是什么东西定义一个段的长度呢？没什么东西，真的——这里，我们将进入一些真正棘手的语义里。一个段与其说是一个位置，不如说是一个界限。一旦定义了一个段从何处开始，该段就可以包括从起始位置到这个界限之间的任何内存位置——那是沿线的 65 536 个字节。

当然，没有东西命令一个段必须使用所有那部分内存。大多数情况下，当一个段在某个段地址被定义时，程序只把之后的几百个或者也许几千个字节看成是这个段的一部分，除非它是一个真正意义上的国际水平程序。当阅读关于段的相关知识时，大多数初学者把它们看作某种类型的内存位置，一种预留出来用于某些特定应用的、两侧都有墙壁的受保护内存区域。

像你可能出到的结论一样，这种观点完全不正确。在实模式下，段内的任何内容都不受保护，而且段并没有被预留出来供某些特定的寄存器或访问方式使用。段也可以重叠。(人们往往从来不去想或者认识到这一点。) 毫无疑问，除了作为某些特定类型的内存引用不能逾越的界限之外，段并不真的存在。再回到 CPU 佩戴的 64KB“眼罩”，像我在图 4-3 中画的一样。我的观点是这样的：段是内存中安装 CPU 的 64KB 眼罩的位置。通过这个眼罩查看内存，能够看到从段地址开始，一直到眼罩阻止你的沿途所有字节，一共 64KB。

段的定义被公认为形而上学，理解它的关键是知道如何使用段——而要想理解其工作原理，最终还需要对一些寄存器的相关知识进行详细探讨。

4.2.2 用两个 16 位寄存器构成 20 位地址

正如我在前几章中非正式提到的那样，寄存器是一个位于 CPU 芯片内部的

存储位置，而不是 CPU 外面的内存中的某个位置。8088、8086 和 80286 通常称为 16 位 CPU，因为其内部的寄存器几乎都是 16 位大小。80386 和它 20 年来非常有价值的接班人称为 32 位 CPU，因为其内部的大多数寄存器是 32 位大小。20 世纪中期以来，许多新的 x86 CPU 被设计为 64 位，其寄存器都是 64 位大小。(本章的结尾会有更多关于它的介绍)。x86 CPU 拥有很多的寄存器，它们实际上是非常有趣的“员工”。

寄存器能做很多项工作，但最重要的一项也许就是存放内存中某些重要位置的地址。如果你还记得，8086 和 8088 有 20 个地址引脚，它们的 1M 字节内存(它是我们正在谈论的实模式分段内存)需要 20 位大小的地址。

如何把一个 20 位的内存地址放到一个 16 位的寄存器中呢？这根本无法做到。

但是，可以把一个 20 位的地址放入两个 16 的位寄存器中。

发生的事情是：所有位于实模式下的 1MB 内存中的存储位置不是只有一个地址，而是两个。内存中的每个字节都假设位于某个段中。那么，一个字节的完整地址包含它所在段的地址，还有该字节距离那个段的起始位置的字节数。回想一下，那个段的地址是该字节的段地址，该字节距离那个段的起始位置的字节数是它的偏移地址。两个地址必须都指定才能完整描述实模式下整整一兆内存中的任意单个字节的位置。当书写时，段地址在前面，偏移地址在后面。两者使用冒号隔开。以段地址：偏移地址的形式表达的地址通常以十六进制描述。

为了帮助你更加清晰地理解它，我已经将其画在图 4-5 中。一个我们称之为“MyByte”的字节数据存放在内存中用黑色标识出来的位置处。它的地址是由 0001:0019 给出的。这意味着 MyByte 位于 0001H 段内，它距离该段的起始位置 0019H 个字节。x86 编程公约规定，当两个数字之间用冒号隔开，用于指定一个地址时，不用再为这两个十六进制数的结尾加上字母 H。以段地址:偏移地址的形式书写的地址被假定为以十六进制表示。

然而，宇宙经常有悖常理，聪明的眼睛会觉察到：MyByte 还可以有另外两个完全合法的地址：0:0029 和 0002:0009。为什么会这样呢？请记住，一个段可能开始于实内存中所有 1M 字节里的任意一个 16 字节处。一旦开始，一个段就包含从它的起始位置一直到 65 535 个字节处的所有内存字节。段重叠也没有什么不对。在图 4-5 中有三个相互重叠的段。

我们可以认为 MyByte 位于距离第一个段(该段的段地址为 0000H)起始位置 2DH 个字节的位置处。我们也可以认为 MyByte 位于距离第二个段(该段的段地址

为 0001H)起始位置 1DH 个字节的位置处。这并不是说 MyByte 一次位于两个或三个不同的位置。它位于一个位置，但是那个位置可以用三种方式之一来表示。

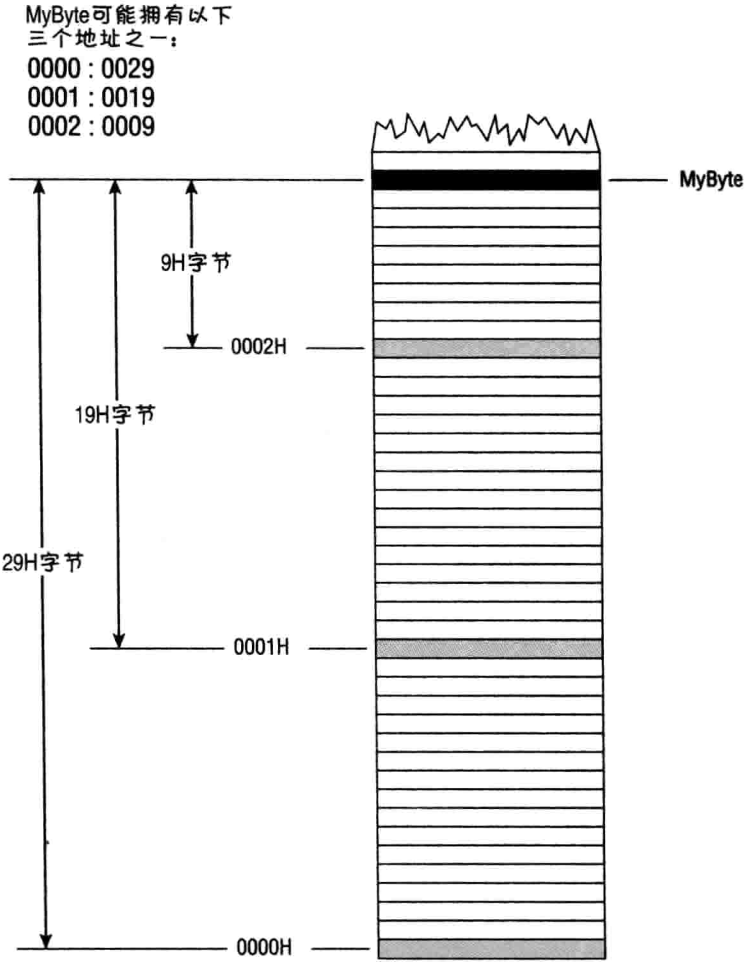


图 4-5 段地址和偏移地址

这有点像芝加哥的街道编号系统。Howard 大街是 Madison 大街(芝加哥的“起点”)北边的第 76 条街，Howard 大街也是 Touhy 大道北边的第四条街道。可以用与 Madison 大街或者 Touhy 大道的相对位置来描述 Howard 大街的位置，取决于你想采用哪一种。

位于实模式下 1MB 内存中间某处的任意一个字节可能属于上千个不同的

段。严格来说，它实际上位于哪个段只是一个习惯问题。

总之，用两个 16 位寄存器表示一个 20 位地址，就是把段地址放入一个 16 位寄存器，把偏移地址放入另一个 16 位寄存器。两个寄存器一起，共同标识实模式 1MB 内存中 1 048 576 个字节中的一个字节。

这很尴尬？的确，但它是多年以来我们能够做到的最佳方式。

4.3 16 位和 32 位寄存器

段地址可被看作实模式下的 64KB 眼罩的起始位置。通常情况下，你会移动眼罩以找到希望工作的位置，然后把眼罩留在一个地方，在它周围的 64K 限制以内的空间范围移动。

这正是寄存器在实模式段模型下汇编语言中的用法。在 8088、8086 和 80286 中，确切来讲，有 4 个专门设计用于存放段地址的段寄存器。386 及后来的处理器又多了两个，它们也可在实模式下使用。(如果打算使用这两个额外的段寄存器，你需要知道正在运行的 CPU 的型号，因为老式的 CPU 中根本没有它们。)每个段寄存器都是一个位于 CPU 芯片本身内部的 16 位内存位置。不管 CPU 正在做什么，只要它对内存中的某一位置进行寻址，那么该位置的段地址就会出现在 6 个段寄存器之一中。

段寄存器拥有反映其一般功能的名字：CS、DS、SS、ES、FS 和 GS。FS 和 GS 只出现在 386 及更高版本的 Intel x86 CPU 中，但大小仍然为 16 位。所有段寄存器的大小都是 16 位，无论 CPU 是什么型号,甚至对 32 位 CPU 来说也是如此。

- *CS*(code segment)代表代码段。机器指令存放在代码段的某些偏移位置处。CS 中包含当前执行指令的代码段的段地址。
- *DS*(data segment)代表数据段。变量和其他数据存放在数据段的某些偏移位置处。也许会有很多个数据段，但是 CPU 一次只能使用一个，这是通过将该段的段地址放入寄存器 DS 来实现的。
- *SS*(stack segment)代表堆栈段。堆栈是 CPU 用来暂时存放数据和地址的一个非常重要的部件。至于堆栈如何工作，我将在稍后进行介绍，现在只需明白，与实模式的 1MB 内存中的所有其他内容一样，堆栈也有一个段地址，这个地址存放在 SS 中。
- *ES*(extra segment)代表附加段寄存器。确切地讲，附加段寄存器是一

个可用于指定内存中某一位置的备用段。

- *FS* 和 *GS* 是 *ES* 的克隆。它们都是额外的段，没有特殊的工作或专长。它们的名字来自这样的事实：它们创建于 ES 之后(想想，E，F，G)。不要忘记，它们只存在于 386 及后来的 x86 CPU 中！

4.3.1 通用寄存器

段寄存器存在的目的只是为了保存段地址。在实模式下，我们也可强制它们做极少数其他事情，但总的来说，段寄存器应该被看成是存放段地址的专家。x86 处理器拥有很多个通用寄存器，它们用来做汇编语言计算的其余工作。在很多(除了计算之外的)其他事情中，这些通用寄存器用来存放偏移地址，它必须与段地址成对出现，以标识内存中的某个位置。在移位操作(详见后文)和许多其他操作中，它们还经常用于存放算术运算值。它们才是 CPU 内部名副其实的“工匠口袋”。

但是，这里让我们来看看较旧的 16 位 x86 处理器(8086、8088 和 80286)和从 386 开始的较新的 32 位 x86 处理器之间最大、最明显的区别之一：通用寄存器的大小。当我于 1989 年编写本书的第 1 版时，8088 仍然统治着个人电脑的计算世界，我把自己限制在讨论 8088 所拥有的特性之中。

那些日子早已过去很久了。完全 32 位的 386 现在已经被人们认为是老古董，随着时间的流逝，早期的 1993 年奔腾系列也愈发显得古怪。现在是 32 位的世界，总有一天它会成为 64 位的世界。“位处理”世界几乎完全由 x86 CPU 的寄存器的宽度来定义。

像段寄存器一样，通用寄存器也是 CPU 芯片本身内部的存储位置；也像段寄存器一样，它们都拥有自己的名字，而不是数字地址。通用寄存器真的是通才，因为所有它们共享很多本领。但是，某些通用寄存器也拥有我所谓的“隐形议程”：只有它才能执行的一个任务或者任务集。我仍沿袭以前的风格解释这些隐形议程——记住，有些隐性议程实际上只是在老式的 16 位 CPU 中才会受到限制。较新的通用寄存器，嗯，要更加通用些。

在当前的 32 位世界中，通用寄存器分为三个一般类：16 位通用寄存器、32 位扩展通用寄存器和 8 位的半寄存器。这三类并不代表三个完全不同的寄存器组。16 位和 8 位寄存器实际上是 32 位寄存器内部的一块区域的名字。x86 CPU 系列产品通过扩展存在于老式 CPU 中的寄存器来实现增大寄存器的容量。在家里添加一个房间，并不能让你拥有两套房子，只是有了一个更大的房子。x86 寄

存器就是这样。

有 8 个 16 位通用寄存器：AX、 BX、CX、 DX、 BP、 SI、 DI 和 SP。(SP 不如其他通用寄存器通用性强，但我们会接触到它。) 这些寄存器都存在于 8086、8088 和 80286 的 CPU 内部。它们的大小都为 16 位，任意能用 16 位或者更少位表示的值都可放入其中。当 Intel 于 1986 年将其 x86 体系结构扩展为 32 位时，它把所有 8 个寄存器的大小增加了一倍，并且通过在原有的名字前面加上前缀 E 来对它们进行新的命名，形成了 EAX、EBX、ECX、EDX、EBP、 ESI、EDI 和 ESP。

那么，这些只是在原有基础上变大了一些的寄存器，还是新的寄存器呢？

都是。

像很多汇编语言中的知识一样，图示比讲解会更清晰。图 4-6 展示了 SI、DI、BP 和 SP 如何将它们的尺寸变为原来的两倍，并获取新的名字——旧的东西一点儿也没丢。

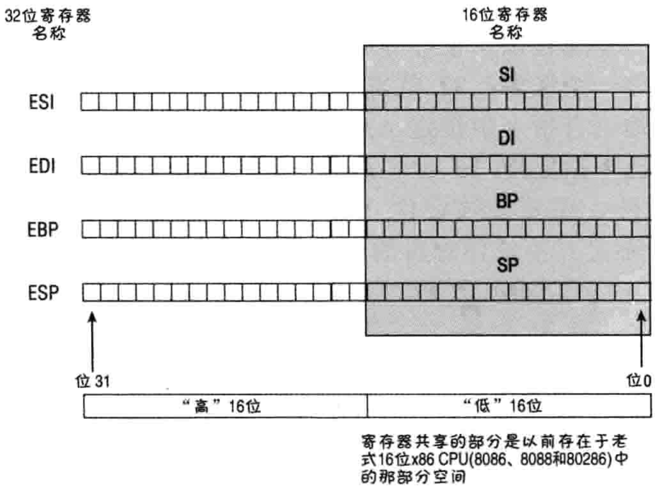


图 4-6　扩展 16 位通用寄存器

图 4-6 中所示的 4 个寄存器都完全是 32 位大小。然而，在每个寄存器里，低 16 位都拥有自己的名字。例如，ESI 的较低的 16 位可以 SI 来进行引用。EDI 较低的 16 位可以通过 DI 来引用。如果正在一台 8088 机器(如古老的 IBM PC 机)上编写运行于实模式下的程序，你只能引用 DI 部分——因为在那款 CPU 中根本不存在高 16 位。

非常不幸的是，32 位通用寄存器的高 16 位根本没有自己的名字。可以通过

SI 来访问 ESI 的低 16 位，但是要想获取高 16 位，必须引用 ESI，获取整个 32 位数据。

4.3.2 半寄存器

其他 4 个通用寄存器(EAX、 EBX、ECX 和 EDX)也一样，但是有一个额外的技巧：低 16 位本身被分成两个 8 位的半寄存器，所以，我们不是拥有两个级别，而是拥有三个级别的寄存器名。16 位寄存器 AX、BX、CX 和 DX 作为 EAX、EBX、ECX 和 EDX 的低 16 位而存在，而 AX、BX、CX 和 DX 本身又被分成 8 位的半寄存器，编译程序能够识别这两个“半寄存器”的特殊名字。字母 A、B、C 和 D 仍然保留，但是，半寄存器不再使用 X，而是用一个 H(高半寄存器) 或一个 L(低半寄存器)来代替它。每个半寄存器的大小都是一个字节(8 位)。因此，字节大小的半寄存器 AH 和 AL 构成了 16 位寄存器 AX；在 BX 中有 BH 和 BL 两个“半寄存器”，依此类推。

再次的，这一点通过图示很好理解(参见图 4-7)。正如前面提到的那样，这一非常有用的系统有一个怪癖：32 位寄存器的高 16 位部分没有名字。换句话说，你可以通过在汇编语言指令中指定 AX 来读取 EAX 的低 16 位，但是没有办法来通过高 16 位自己来指定高 16 位。这使得寄存器命名规范稍微简单一些，(除了所有其他内容之外，还要必须记住 EAXH、EBXH、ECXH 和 EDXH，你愿意吗？)但是，这一不足并不像你想象的那样经常觉察得到。

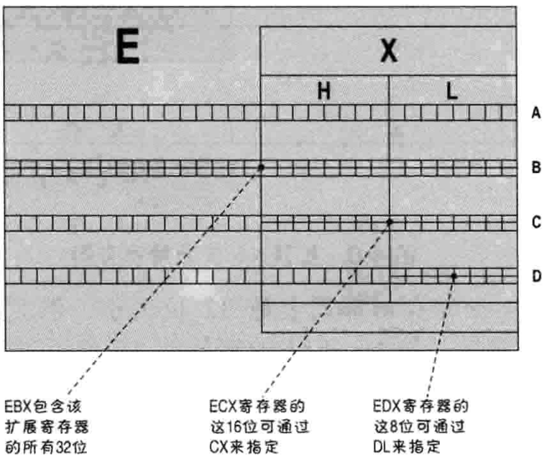


图 4-7　8 位、16 位和 32 位寄存器

8位“半寄存器”的好处是可以读取和修改16位数据的其中一半，而不影响另一半。这意味着，如果将一个字大小的十六进制值76E9H放入寄存器AX，你可以从AH中读取字节大小的值76H，从AL中读取0E9H 。更妙的是，如果把0AH存入寄存器AL，然后再次读出AX寄存器，你将发现，原来的值76E9H已经变为760AH。

当处理很多8位量时，能够将AX、BX、CX和DX寄存器看成8位的半寄存器会带来很大的方便。每个寄存器的一半都可以被视为一个单独的寄存器，在工作时，可以提供两倍数量的地方来存放数据。在紧要关头找到一个地方来保存一个值是汇编语言程序员经常面临的巨大挑战之一，这一点你以后将会看到。

注意，这一双重特性只涉及16位通用寄存器AX、BX、CX和DX。其他的16位通用寄存器(SP、BP、SI和DI)并不具有类似的特性。例如，不存在名为SIH和SIL的像在AX、BX、CX、DX中一样方便的8位寄存器。

4.3.3 指令指针寄存器

x86处理器里面还有一种类型的寄存器。*指令指针寄存器*(instruction pointer，通常称为IP，或在32位保护模式下，EIP)自成一派。与8个通用寄存器形成鲜明对比的是，IP是一个出类拔萃的专家——与其说它是段寄存器，不如说它是一个专家。它只可以做一件事情：包含当前代码段中下一条即将执行的机器指令的偏移地址

代码段(code segment)是一块存储机器指令的内存区域。程序执行的测试和步骤就包含在代码段中。根据正在使用的编程模型(稍后会有详细介绍)，一个程序可能有很多个代码段或者只有一个。当前代码段(current code segment)是指段地址当前正存放在代码段寄存器CS中的那段代码。在任意给定时间里，当前正在执行的指令都出现在当前代码段中。在实模式段模型里，CS中的值经常变化。在两个平面模型里，CS中的值(几乎)从来没有变化——当然也从来没有改变对应用程序的“叫牌”。(正如稍后你将看到的那样，在保护模式下的所有段寄存器都“隶属于”操作系统，一般的应用程序不能修改它。)

当执行程序时，CPU使用IP来跟踪位于当前代码段中的位置。每执行一条指令后，IP就递增一定数量的字节。这个字节数是刚刚执行的那条指令的大小。最终的结果是IP跳到了内存中更远的地方，指向下一条即将执行的指令的开始位置。指令有不同的大小，典型范围是1到6个字节。(其中某些更神秘的指令的更神秘的形式可能更大些。) CPU谨慎地将IP增加正确数量的字节数，因此，

它实际上最终指向下一条指令的起始位置，而不是指向最后一条指令或者其他指令的中间位置。

如果 IP 包含下一条机器指令的偏移地址，那么段地址在什么地方呢？段地址存放在代码段寄存器 CS 中。CS 和 IP 一起，保存下一条即将执行的指令的完整地址。

地址的本质取决于使用的 CPU，以及正在使用的针对该 CPU 的编程模型。在 8088、8086 和 80286 中，IP 的大小是 16 位。在 386 及更高的 CPU 中，IP(像除段寄存器之外的其他所有段寄存器一样)增加到 32 位的规模，变成 EIP。

在实模式段模型中，CS 和 IP 一起工作，带来一个 20 位的地址，指向实模式内存中的 1 048 576 个字节之一。

在两种平面模型中(稍后会详细介绍)，CS 由操作系统设置并且保持不变。IP 完成你(程序员)必须处理的所有指令指向。在 16 位平面模型中(实模式平面模型)，这意味着 IP 可以跟踪所有 64K 内存段内的指令执行。32 位平面模型做的远远超过其一倍，32 位可以代表 4 294 967 296 个不同的内存地址。因此，在 32 位平面模型(即，保护模式平面模型)中，IP 可以跟踪 4GB 内存范围内的指令执行——这是一个在以前简直难以想象的内存容量，现在已经相当常见了。

值得注意的是，IP 是唯一既不能直接读出，也不能直接写入的寄存器。有一些技巧可用来获取当前 IP 寄存器的值，但是拥有 IP 的值并不是像你想象得那么有用，所以，你不会经常这样做。

4.3.4 标志寄存器

在 CPU 的内部，有一种额外类型的寄存器，即所谓的标志寄存器(flags register)。在 8086、8088 和 80286 中，它的大小为 16 位，并且它的正式名称是 FLAGS。在 386 及更高的 CPU 中，它的大小是 32 位，并且它在 32 位 CPU 中的正式名称为 EFLAGS。标志寄存器中的绝大部分位都可作为名为*标志*位的“1 位”寄存器来使用。这些单个标志位都有一个名字，例如 CF、DF、OF 等，并且每一个在 CPU 内部都有一个特定的含义。

当程序执行测试时，它所测试的就是标志寄存器中的某个单个的标志位。因为每个位可以包含的值只有两个：1 或者 0，所以，在汇编语言中，测试确实是二选一的：标志位被设置为 1，或者没有被设置为 1。如果某一标志被设置为 1，程序执行一个行动；如果该标志被设置为 0，程序采取另外一个不同的行动。

标志寄存器几乎不会作为一个单位来处理。具体情况是，许多不同的机器指

令测试各种标志位，以决定在那些非此即彼的决策中，到底走哪一条路。我们目前正专注于内存寻址的学习，所以现在我承诺，本书后面部分将在合适的时机(即当我们讨论测试标志寄存器里的各种标志位的机器指令时)对标志寄存器的相关知识进行更加详细的介绍。

4.4　三种主要的汇编编程模型

我在本章前半部分提到，x86 处理器可以使用三种主要的编程模型，但其中有两个现在已被认为是过时的。它们之间的区别(主要)在于进行内存寻址时对寄存器的使用。(其他方面的差异，尤其是高端部分，大多被操作系统隐藏起来。)本节将介绍这三种模式，在本书其余部分课程中，我们将遇到每一种模式。

4.4.1　实模式平面模型

在实模式下，如果你还记得，CPU 可以看到的只有 1MB(1 048 576)的内存。通过使用前面讲过的段地址：偏移地址技巧，用存放在两个寄存器中的两个 16 位地址形成 20 位地址，你可以访问这百万余个字节中的每一个字节，或者，根本不受段的愚弄，干脆使用 64KB 内存。

在实模式平面模型下，程序和它处理的所有数据都必须位于一个 64KB 大小的内存块中。64KB！老天！只有 64K 个字节，你能做什么？嗯，为 IBM PC 机编写的第一个版本的 WordStar 适合于 64KB。Turbo Pascal 的前三个主要发布版本也适用于 64KB——实际上，Turbo Pascal 程序本身所占用的空间远少于 64KB，因为它将由它编写的程序也编译到内存中。整个 Turbo Pascal 程序包——编译器、文本编辑器和一些奇怪的工具——刚刚超过 39KB。39KB！如今那么小的空间甚至连(使用 Microsoft Word)给妈妈写封信都不够！

是的，是的。但是，那主要是因为我们不必非要那么做。内存已经变得相当便宜，以历史标准衡量，我们现在的计算机包含惊人数量的内存。我们变得懒惰、贪婪和浪费。

在 64KB 的内存空间里，曾经发生过十分壮观的事情，虽然你可能永远不会呼吁把自己限制在实模式平面模型里，但是，那些头发花白的老程序员们为它制定的所有规矩都非常有用。更重要的是，实模式平面模型是保护模式平面模型的“小兄弟”，而后者将是你在 Linux 下编程所使用的代码模型。如果学会了实模式平面模型，保护模式平面模型将是小菜一碟。(你将遇到的任何麻烦都不会来自

汇编代码或存储器模型，而是来自拜占庭式的 Linux 请求和它的规范的代码库。)

我已经通过图表将实模式平面模型显示在图 4-8 中。它无须过多地描述。所有的段寄存器都被设置为指向你能使用的 64KB 内存块的起始位置。(当加载并运行你的程序时，操作系统会设置这些段寄存器。)只要你的程序运行，它们就都指向同一个地方，而且永远也不会改变。在这种情况下，你可以完全忘记它们。噢！没有段寄存器，没有段的愚弄，没有它们带来的“丑陋的”并发症。

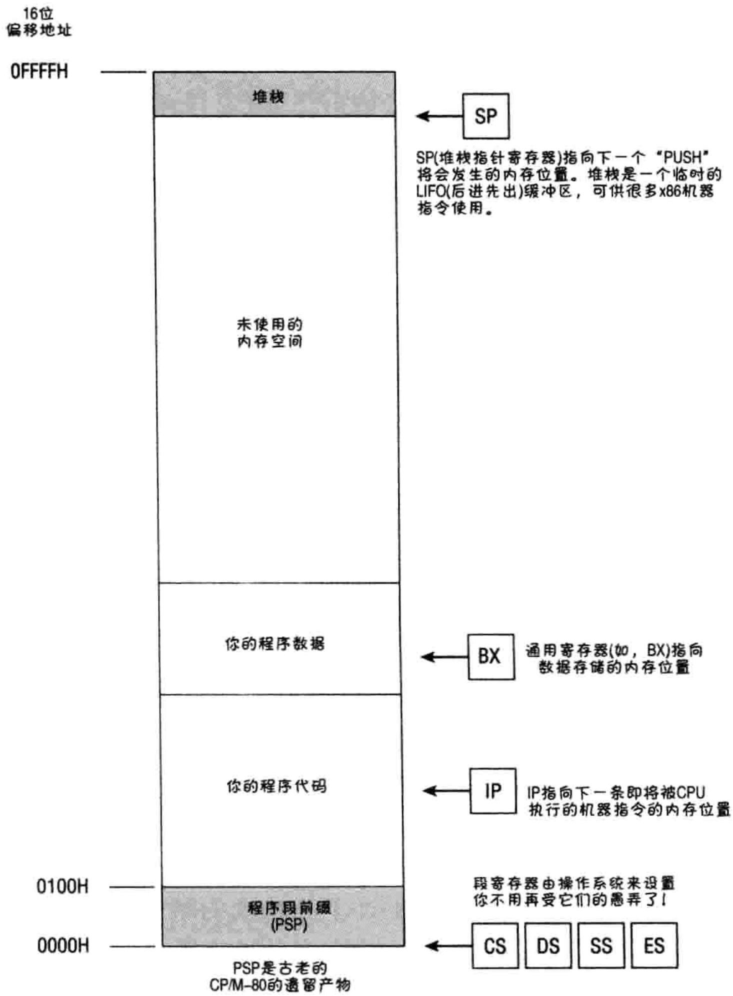


图 4-8　实模式平面模型

由于 16 位寄存器(如 BX)可以容纳从 0 到 65 535 之间的任意值，所以它能够精确定位程序使用的 64KB 空间中的任一字节。因此，内存寻址可在不必显式使用段寄存器的情况下完成。当然，从 CPU 的角度来看，段寄存器仍然在发挥作用。段寄存器并没有消失，它们依然在那里，只是当加载你的程序时，操作系统会根据自己的选择来设置它们的值，而只要你的程序运行，这些值就会“乖乖地听话”。在编写程序时，你不必以任何方式访问段寄存器。

大多数通用寄存器可以存放内存中的存储位置。可以通过将它们与机器指令相结合，来实现从内存中取出数据，以及将数据写回内存。

在程序所在的段的顶部，会看到一块名为堆栈(stack)的小区域。堆栈是 LIFO(后进先出)的存储位置，它具有一些非常特殊的应用。我在后面的部分会对堆栈的概念及其工作机制进行详细的讲解。

4.4.2 实模式段模型

这本书的前两个版本完全侧重于实模式段模型。这是整个 MS－DOS 时代的主流编程模型，当你启动 MS－DOS 窗口，运行旧的“遗留问题”软件时，它仍然在发挥作用。这是一个复杂而丑陋的系统，要求你必须记住很多小规则和很多“陷阱”，但是理解它非常有用，因为它非常清晰地阐明了段的本质和功能。请注意，在两个平面模型下，你都可以闭上眼睛，假装段和段寄存器并不真正存在，但是它们不仅存在而且在工作，一旦陷入一些具有更多外来风格的编程，你将需要意识到它们的存在，并且掌握它们的工作原理。

在实模式段模型下，程序可以看到在实模式下 CPU 可用的整个 1MB 内存。它通过把 16 位的段地址和 16 位的偏移地址结合在一起来做到这一点。但是，它不是简单地将它们合在一起构成 32 位地址。回想一下本章前面关于段的讨论。段地址并不是一个真正的内存地址。段地址指定了段到底从 65,536 个可用插槽中的哪一个开始。从内存底部到顶端每 16 字节都有一个这种插槽。段地址 0000H 指定了第一个这样的插槽，它在内存中的第一个位置。段地址 0001H 指定了下一个插槽，它位于比上一个插槽高 16 个字节的内存处。再往上跳 16 个字节，将得到段地址 0002H，以此类推。可以通过把一个段地址乘以 16 来将其翻译成一个真正的 20 位内存地址。因此，段地址 0002H 相当于内存地址 0020H，这是内存中的第 32 个字节。

但是，这种乘法并不需要你来做。CPU 会在内部将段地址和偏移地址组合成一个完整的 20 位地址。你的工作是告诉 CPU 那个 20 位地址的两个不同组成部

分位于什么地方。惯用的表示法是在段寄存器和偏移寄存器之间用一个冒号隔开，如下例所示：

```
SS:SP
SS:BP
ES:DI
DS:SI
CS:BX
```

以上 5 个寄存器组合中的每一个都可以指定一个完整的 20 位地址。例如，ES:DI 指定了一个地址，该地址距离 ES 段的起始位置 DI 个字节。

我已经在图 4-9 中通过一幅简图来勾勒实模式段模型。与实模式平面模型(见图 4-8)相比，这里的图显示了所有的内存，而不是只有实模式平面模型程序运行时分配的一个小小的 64KB 块。为实模式段模型编写的程序可以看到所有的实模式内存。

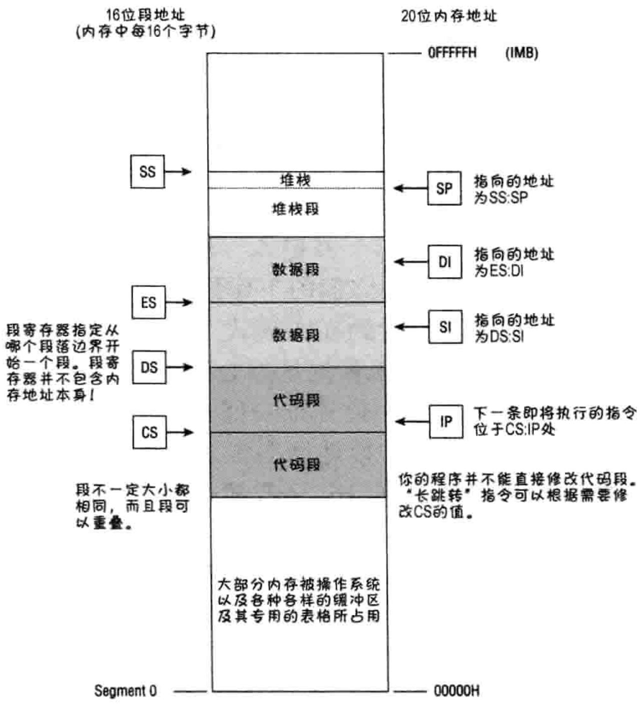


图 4-9 实模式分段模型

该图显示了两个代码段和两个数据段。在实践中，可以有任意合理数量的代码段和数据段，每种数量不止 2 个。你可以在同一时间访问两个数据段，因为有两个段寄存器可以做这项工作：DS 和 ES。(在 386 及更高版本的处理器中，还有两个附加段寄存器：FS 和 GS。) 每个都可以指定一个数据段，并且可以通过使用机器指令将数据从一个段移到另一个段中。但是，你只有一个代码段寄存器：CS。CS 总是指向当前的代码段，IP 寄存器指向下一条即将执行的指令的地址。不能直接加载一个值到 CS 中来实现从一个代码段到另外一个代码段的改变。如果必要的话，名为跳转的机器指令可以实现到另外一个代码段的改变。你的程序可以跨越几个代码段，当一个跳转指令(有好多种跳转指令)需要执行到另外一个段的跳转时，它将为你改变 CS 中的值。

任意一个单独的程序都只有一个堆栈段，它由堆栈段寄存器 SS 来指定。堆栈指针寄存器 SP 指向下一条堆栈操作即将发生的(相对于 SS 而言，方向刚好是颠倒的)内存地址。对于堆栈，仍然需要进一步的详细讲解，我将在本书后面几个地方给出。

必须记住，在实模式下，也许有几个操作系统(如果你使用的是 8086 或 8088，那将是整个操作系统)与你的程序一起存在于内存之中，同时还有一些重要的系统数据表。你可能由于粗心使用段寄存器而破坏部分操作系统，这将导致操作系统崩溃，并导致你的程序与之一起崩溃。正是这一危险促使英特尔在它的 80386 及更新的 CPU 上建立新的属性来支持“保护”模式。在保护模式下，应用程序(也就是你写的程序，与操作系统或设备驱动程序相对)不能破坏操作系统或其他通过多任务技术碰巧在内存中运行的应用程序。

最后，在 80286 中，有一种初步的保护模式，尽管这是事实，但从未有操作系统真正使用过它，它如今已经没有讨论价值。

4.4.3　保护模式平面模型

自从 1986 年 386 出现开始，Intel 的 CPU 已经实现了很好的保护模式体系结构。但是，应用程序只通过它们本身并不能使用保护模式。在应用程序能够在其内部运行之前，操作系统必须建立和管理一个保护模式。MS-DOS 做不到这一点，微软的 Windows 也不能真正做到这一点，直到 Windows NT 首次出现于 1994 年。 在 Linux 中，没有实模式“遗留问题”需要处理，自从 1992 年第一次出现以来，它就一直运行在保护模式下。

我们可以为 Linux 和 NT 之后发布的 Windows 编写保护模式汇编语言程序。

(由于技术原因，我排除了 Windows 9x。它的内存模型是一个奇怪的实模式和保护模式的混合体，并且很难完全理解——现在已经完全不符合实际需要了。)应该注意的是，为 Windows 编写的程序本质上不需要图形化。在 Windows 操作系统下，编写保护模式程序的最简单方法是创建控制台(console)应用程序，它是文本模式程序，运行在一个名为控制台的文本模式窗口下。可通过几乎与 MS-DOS 下完全相同的命令行来控制控制台。控制台应用程序使用保护模式平面模型，与编写 Windows GUI 应用程序相比，相当简单。Linux 的默认模式是文本控制台，所以为 Linux 创建汇编程序更加容易，更多的人似乎已经在这样做。内存模型大同小异。

我已经把保护模式平面模型画在图 4-10 中。你的程序看到的是一块地址从 0 一直到 4G-1 的内存。每个地址都是一个 32 位的量。所有通用寄存器的大小都是 32 位，因此一个通用寄存器可以指向整个 4GB 地址空间中的任何位置。指令指针寄存器的大小也是 32 位，所以 EIP 可以指向 4GB 内存空间内的任何地方的任何机器指令。

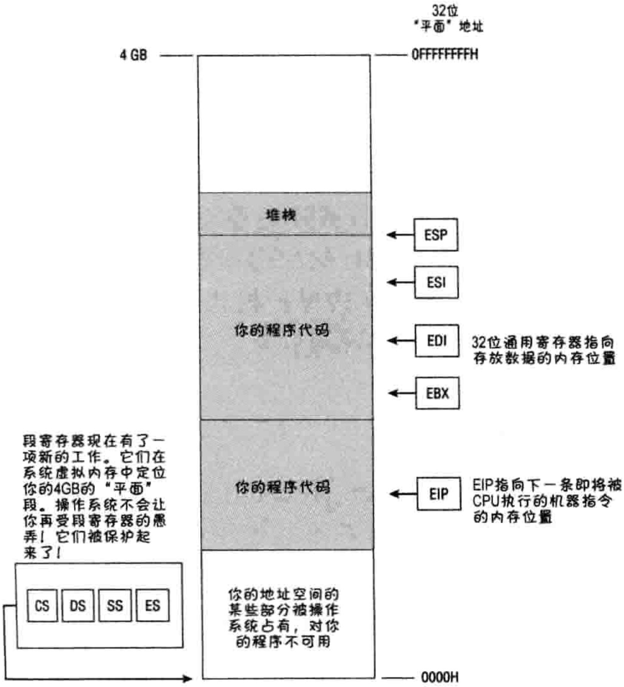


图 4-10　保护模式平面模型

段寄存器仍然存在，但是它们以一种完全不同的方式工作。不是你不一定受它们愚弄，而是你已经不能受它们愚弄。现在，段寄存器被认为是操作系统的一部分，在几乎所有情况下，你既不能直接读取它们，也不能直接修改它们。它们的新工作是定义你的 4GB 内存空间出现在物理内存或者虚拟内存的什么地方。物理内存可能远大于 4GB，而且目前 4GB 的内存并不特别昂贵。然而，一个 32 位寄存器只能表示 4 294 967 296 个不同的位置。如果你的计算机拥有超过 4GB 的内存，操作系统必须在内存中安排出 4GB 区域，你的程序仅限于操纵该区域内的内存。在较大的内存系统中定义 4GB 区域的范围，这就是段寄存器的工作，而且操作系统让它们寸步不离其“防护衣”。

在本书中，我不会对虚拟内存做过多的讲解。它是一个系统，可以根据它将一个更大的内存空间“映射”到磁盘存储器中，因此即使你的计算机中只有 4GB 的物理内存，CPU 也能够处理一个数以百万计字节的、更大空间的“虚拟内存”。这也是由操作系统来处理的，而其处理方式对你编写的软件几乎完全透明。

当程序运行时，它将得到一个 4GB 的地址空间来施展手脚，任何一个 32 位寄存器完全依靠自己就可以访问这四十亿内存中的任一位置，理解这些就足够了。这是一个过于简化的描述，尤其是对于普通的基于 Intel 处理器的台式电脑而言。并非所有的 4GB 内存你的程序都可以支配，有些内存空间你不但不能使用，甚至连看都不能看。不幸的是，运行的操作系统不同，具体的规则也有所不同，在没有指定 Linux 或 Windows NT 或其他一些保护模式的操作系统之前，我不能一概而论。

但是，回过头看一下图 4-8，比较一下实模式平面模型和保护模式平面模型，还是很有价值的。两者的主要区别是：在实模式平面模型下，你的程序拥有操作系统交给给它的整个 64KB 内存空间。在保护模式平面模型下，你将得到分配给自己的 4GB 内存中的一部分，而其他部分仍然属于操作系统。除此之外，二者有着惊人的相似之处：通用(GP)寄存器本身可以指定整个内存地址空间中的任一内存位置，段寄存器真的是操作系统的工具，而不是你(程序员)的工具。(同样，在保护模式平面模型下，一个通用寄存器可以存放其 4GB 空间内的任意位置的地址，但是当真正读取或写入某些位置时，会被操作系统禁止并且触发运行时错误。)

请注意，我们还没有真正详细地介绍机器指令，但是已经能够非常简洁地定

义机器指令存在和工作的世界了。内存寻址和寄存器都是业务的关键。如果了解了它们，指令将是小事一桩。如果不了解它们，指令不会有任何用处！

保护模式平面模型下编程的困难在于了解操作系统及其要求和限制。这可能需要大量的学习：Windows NT 和 Linux 是主要的操作系统，要想深入了解它们，可能需要几年的研究。在本书中，我将引入在平面模型下的保护模式汇编编程，但你必须自学操作系统。这本书仅仅是个开始，你还有很长的路要走，而现在你几乎连路边还没有挨到。

4.5　保护模式下不再允许我们做的事情

如果阅读本书的人们具有一些 DOS 知识，他们可能还记得，在 DOS 环境下，整个计算机“敞开”供 DOS 程序访问。当然，DOS 是一个 16 位操作系统，只能访问低于 1MB 的内存地址空间。然而，这一空间可以驻留很多有趣的事情，它们和加载一个地址到 DS:AX 带来的乐趣不相上下。那些日子早已经过去了，我们的境况也因此而变得好起来，但是在执行很多有用的功能时，有一种令人陶醉的简单，我承认，有些功能我已经忘记了。这种简单性使得讲解基本汇编语言技术容易了很多，这一点看过本书前几版的人可能深有同感。

理解保护模式下不再允许我们做的事情是非常有用的，尤其是对像我一样的 DOS 时代程序员而言。

4.5.1　内存映射视频系统

最初的 IBM 个人电脑使用一个很简单的、非常巧妙的机制来显示文字和(按今天的标准衡量)低分辨率的图形。显卡包含一定数量的内存，这块内存“映射”到个人电脑的物理内存空间。换句话说，在访问显卡内存过程中并没有什么“魔术”参与。简单地将数据写到由段地址：偏移地址指定的、位于显卡包含的内存范围内的某一地址，就可以将一些内容显示在显示器上。

这种技术允许程序显示满满一屏的文本，这些文本将一下子弹入我们的视野，没有任何文字从上到下逐渐出现的感觉，即使是在早期那些拥有慢得令人难以置信的 CPU 芯片的机器上。内存缓冲区的组织非常简单：从地址 0B00:0(或彩色显示器上 0B800:0)开始，是一个两字节字的数组。每个字的第一个字节是一个字符的 ASCII 编码。例如，数字 41H 是大写字母“A”的编码。第二个字节是一

个文本属性：字形的颜色、字符单元格的背景颜色，或者诸如下划线之类的特殊表示方法。

这种安排使得使用相对简单的汇编语言库能够十分容易和快捷地显示文本。不幸的是，这种直接访问系统外设的行为是有违保护模式下的保护条款的。原因很简单：保护模式可以让多个程序在同一时间执行，如果不止一个正在执行的程序企图同时改变显存，将会导致视频混乱。DOS 是严格的单任务操作系统，因此无论何时都只能有一个程序在运行。

为了能以一种合理的方式拥有多任务处理功能，操作系统不得不通过精心制作的视频显示代码库来“管理”视频，依次地，这些代码库通过与内核一同位于内核空间的驱动程序访问显示硬件。驱动程序使得操作系统能够限制单个程序到屏幕窗口的视频输出，以便任意数量的正在运行的程序可以同时显示其输出，而不必和所有其他程序的输出发生碰撞。

综上所述，有一种方法能够在用户内存中建立一个缓冲区，然后再告诉 Linux 把它当作显存来使用。这涉及一些与 Linux 帧缓冲设备 dev/fb0 及函数 mmap 和 ioctl 相关的忙乱工作，但是它还不够简单，也不够迅速。该机制是有用的，因为它能够将古老的 DOS 程序移植至 Linux 操作系统上，但是对于新程序而言，它的麻烦远远超过其价值。在本书后面部分，我将使用一个控制台窗口和 VT100 终端控制序列演示一种颇受青睐的处理文本屏幕输出的 Linux 方法。

4.5.2　直接访问端口硬件

在 DOS 时代，个人电脑拥有由主板上单独的控制器芯片所控制的串行端口和并行端口。与计算机的所有其他组件一样，这些控制器芯片可被运行在 DOS 下的任意软件直接访问。通过编写对芯片的位映射控制值以及创建自定义中断服务程序，人们可以创建自定义“微调”串口软件，这使得当时沉重缓慢的、每秒 300 个字符的拨号调制解调器能够以其可达的最大速度进行工作。这是常规情况，但如果加上一点聪明才智，你可以让标准的计算机硬件做它并不真的打算做的事情。例如，通过研究电脑并行端口的硬件控制器，我可以用汇编语言编写一个双向通信系统，该系统可以通过并行端口，把数据快速地从一台计算机转移到另一台计算机上。(这实际上是以前的个人电脑，使用的是 CP/M 操作系统，Z80 的处理器。)

同样，像视频一样，多任务需求也要求操作系统管理对端口的访问，这可通过驱动程序和代码库来实现，但是与视频不同，使用驱动程序作为端口处理的接

口实际上比完全靠自己控制端口简单得多。

4.5.3 直接调用 BIOS

我们必须向森严的保护模式屈服的第三种 DOS 时代技术是直接调用 PC 机的 BIOS 例程。正如第 3 章中讲过的那样，IBM 公司把一个代码库放到只读存储器上，用于对视频和诸如端口之类的外设进行基本的管理。在 DOS 时代，有可能用软件直接调用这些 BIOS 例程，而且没有任何限制。在本书的以前版本中，我讲解了该过程的实现原理以及与之相关的文本视频管理。

保护模式预留出 BIOS 调用给操作系统，但是说实话，那个时候，即使是保护模式下的操作系统，也很少直接调用 BIOS。几乎所有的低级硬件访问都是通过可安装的驱动程序来完成的。操作系统大多利用 BIOS 调用来确定诸如电源管理之类的事情的硬件配置信息。

作为一个安慰奖，Linux 提供了一个低级函数列表，可通过一个与 BIOS 调用非常相似的机制来进行调用，即使用软件中断 80H。我将在本书稍后部分讲解什么是软件中断，以及如何使用它们。

4.6 展望未来: 64 位“长模式”

未来已与我们相伴，你可以在电子公司买到它。除了最便宜的之外，当今所有的个人电脑都使用 AMD 或 Intel 处理器，它们在技术上已经达到了 64 位宽。为了使用这些 64 位特性，需要一个明确为它们编制的操作系统，并且知道如何管理它们。Windows 和 Linux 都有为 64 位“长模式”编译的可用版本。Windows Vista 和 Windows XP 的可用 64 位版本也已经存在一段时间了。Windows 7(我们最好知道)既有 32 位版本，也有 64 位版本。对于 Windows 和 Ubuntu Linux 操作系统，你都必须选择想要的版本。一个“尺码”并不适合所有的情况。在这本书中，我针对的是 Linux 的 32 位版本，并保证，一切程序都将能在 64 位 Linux 的 32 位兼容性模式下运行。然而，知道长模式提供了哪些内容是非常有用的，这样当你的编程技能成熟时，就可以对其进行独自探索了。

64 位 x86 结构有一段特殊的历史：在 2000 年，英特尔的竞争对手 AMD 宣布了 IA－32 体系结构的 64 位超集。AMD 并没有发布执行这种新体系结构的 CPU，直到 2003 年，但是它是 CPU 战争中的一种先发制人的进攻。英特尔已经拥有了 64 位的体系结构，称为 IA－64 安腾，但是安腾是与 IA－32“彻底决

裂”的，如果不进行重新编译，IA－32 软件将无法运行在安腾处理器上，而且在某些情况下，需要重新编码。业界希望向后兼容，他们对 AMD 新体系结构的响应非常积极，Intel 被迫追赶和实现一个 AMD 兼容的体系结构，并将其命名为英特尔 64。Intel 的首款 AMD 兼容的 64 位 CPU 于 2004 年年底发布。现在，大家使用中性供应商术语“x86－64”来描述两家公司实现的相同属性。

x86－64 体系结构定义了三个一般模式：实模式、保护模式和长模式。实模式是一种兼容模式，它使 CPU 能够运行旧的实模式操作系统和软件，如 DOS 和 Windows 3.1。在实模式下，CPU 像 8086 或其他 x86 CPU 在实模式下一样工作，并且支持实模式平面模型和实模式段模型。保护模式也是一种兼容模式，并且使得 CPU 在软件看来就像是一个 IA－32 CPU，以便 x86－64 处理器能够运行 Windows 2000/XP/Vista/7 和其他的 32 位操作系统(如 Linux)，以及它们的 32 位驱动程序和应用程序。

长模式是实 64 位模式；当 CPU 在长模式下，所有的寄存器都是 64 位宽，所有工作于 64 位的机器指令都是可用的。

所有在 IA－32 下可用的寄存器都依然存在，并已被扩大到 64 位宽。64 位版本的寄存器被以 R 开头进行重新命名：EAX 变成 RAX、EBX 变成 RBX，依此类推。除了 IA－32 中我们熟悉的通用寄存器之外，还有 8 个全新的 64 位通用寄存器，它们没有对应的 32 位部分。这些全新的寄存器被命名为 R8 到 R15。我没有谈论太多 x86 架构的快速数学运算特性，也不会在这本书中进行探讨，但 x86－64 增加了 8 个 128 位的 SSE 寄存器到 IA－32 的 8 个同类寄存器中，一共有 16 个 SSE 寄存器。

这些新的寄存器就像是从天上掉下的甘霖，撒到那些寻求更高执行速度的汇编程序员身上。最快的数据存储场所是寄存器，那些饱经早期 x86 CPU 寄存器稀缺之苦的程序员们将看到成堆的内部财富，他们终于可以长吁一口气了。

64 位内存：前景 vs.现在

正如我在前面所说的那样，32 位只能寻址 4 GB 的内存。为了给那些运行在 IA－32 处理器下的程序创造更多的可用内存，我们使用了各种各样的技术。在 64 位长模式下，我们遇到的问题和上述刚好相反：64 位能够寻址如此惊人的巨大内存，以至于需要使用 64 位地址空间的宝贵价值的内存系统即使多年以后也不可能创造出来。(这里，我想通过提醒自己和大家，在此之前我们曾经说过类似的话来回避此类问题，不料却又反复讲到了它。)

64 位可以寻址 16 艾字节。艾字节是 2^{60} 个字节，可用更容易理解的十亿 GB 来形容，这是一个比千万亿字节还多一点的量。我们的电脑硬件有一天将达到那个水平，但是现在还没有。现在这里的“踢球者”是：管理所有这 64 位地址需要 CPU 微架构内部的晶体管。因此，芯片制造商们不是浪费芯片上的晶体管来管理在 CPU(甚至 x86－64 体系结构本身)有生之年都不会用到的内存地址线，而是限制了当前 CPU 实现中的真正功能性地址线的数量。

今天可以买到的 x86－64 CPU 芯片能够实现 48 位虚拟内存地址，但是只能实现 40 位物理内存地址。这比你当前可以塞进物理计算机的任何物理内存还多得多：2^{40} 是 1T 字节，基本上比 1000GB 或 1 万亿字节多一点。我知道的一些高端机器可以访问 64GB 内存。TB 级还得需要几年的时间。

我所说的一切都是为了强调，在现在和未来数年中，为 IA－32 体系结构编程并没有阻止你任何事情。下一章中将要讲述的 NASM 汇编程序可以生成 64 位长模式代码，如果已经安装了 64 位版本的 Linux，你现在可以为它编写代码了。64 位 Linux 在处理函数调用的方式上有些不同，但是 64 位长模式仍然是一个平面模型，它与 32 位平面模型的相似之处，远远超过了 32 位平面模型与愚昧的实模式段模型的相似之处。在前 15 年或者 20 年的个人电脑时代，我们必须忍受实模式段模型。

好了，对于代码即将运行的平台，暂时就讲这么多。现在是时候开始谈谈编写汇编语言程序的过程以及使用的工具了。

第 5 章

汇编的权利

汇编语言程序的创建过程

鲁德亚德·吉卜林(Rudyard Kipling)的诗歌《在新石器时代》(1895)对部落确定性进行了一个漂亮的微讽。在用他那令人信赖的闪长岩战斧成功展示自己之后，诗歌《新石器时代》的叙述者吃掉了他的前敌，并祝贺自己走了一条真正的部落之路。唉，他的图腾柱有其他的想法，并且在午夜的视野里，摆正了我们骄傲的叙述者的位置：

“有 69 种建立部落规定的方法，它们之中的每一个方法都是正确的！”

这首诗的寓意是信任你的图腾柱。对部落规定正确的，同样对编程方法也正确。至少有 69 种编写程序的方法，多年以来我已经尝试了其中大部分。它们互不不同，但都能工作，因为它们都能产生可以加载和运行的程序——只要程序员想出如何遵循某一特定方法，并使用适用于该方法的工具。

不过，尽管所有这些编程方法都能工作，但它们是不能互换的，适用于一种编程语言或工具集的编程方法到了另外一种语言或工具集中也许就不灵了。1977 年，我通过敲入一行一行的代码，然后观察每一行的结果来学习用 APL(A Programming Language 的英文词头，多么深刻)。这就是 APL 的工作方式：每行代码通常是一个独立的实体，它执行计算或者某种类型的数组操作，一旦按下回车键，该行就会产生一个结果并打印出来。(我是在一台电动打印机/终端上学习它的。)当然，你也可以将这些行串在一起，我就是这样做的，但是，从最初的完全无知状态开始创建一个程序，每次一小步，测试所有内容，这是一种令人陶醉的方式。

后来我采用与学习 APL 几乎相同的方式，学会了 BASIC，还有后来的 Perl，但也有一些需要其他技术的其他语言。Pascal 和 C 都需要提前进行大量的学习，因为你再不能像以前一样，敲入一行代码，然后独立执行它们了。这种情

况持续了很久。当 Windows 成为主流之后，Visual Basic 特别是 Delphi 从根本上改变了规则。编程变成了一种刺激反应机制，在该机制中，操作系统发出所谓事件(event)的刺激(敲击按键、鼠标单击等)，简单的程序主要是由对这些事件的反应组成。

汇编语言的组织方式与 C、Java 或者 Pascal 的组织方式不同。最明显的区别是，不能通过试验和错误来编写汇编语言程序，也不能通过让别人替你完成思考来编写汇编语言程序。与 BASIC 或 Perl 或那些诸如 Delphi、Lazarus 或 Gambas 之类的可视化环境相比，它是一个复杂而棘手的过程。一定要注意这一点。你必须阅读“乐谱”。最重要的是，必须要勇于实践。

在本章中，我将按照我自己学习它们的方式，教给你汇编语言的“部落规定”。

5.1　文件及其包含的内容

所有编程过程都是处理文件的过程。有些编程方法隐藏了其中一些文件，所有方法在一定程度上都是为了让人类更加容易地了解这些文件里面的内容，但底线是你要创建文件、处理文件、阅读文件和执行文件。

大多数人都明白，文件是一个存储在某种介质(如硬盘、U 盘或闪存卡、光盘或偶尔某种类型的外来装置)上的数据集合。我们为数据集合赋予一个名字，并将其作为一个单位来进行处理。操作系统控制着对存储介质上的文件的管理。最终，它从数据文件里提出数据，供你查看，并将你所做的更改写回该文件或者某个通过操作系统创建的新文件中。

对于汇编语言而言，有一点是值得注意的，即它在你面前几乎什么都没隐藏；并且要想学好它，你必须愿意进入你所处理的任何文件的内部，通常在字节，不，更多时候是在位的层面上去理解它。这需要更长的时间，但是它会给你带来巨大的知识回报：你将知道一切是如何工作的。相比之下，APL 和 BASIC 则很神秘。我输入了一行代码，然后计算机返回一个响应。之间发生的事情被隐藏得严严实实。在汇编语言中，你将看到所有这一切。诀窍就是理解你正在看的内容。

5.1.1　二进制文件 vs. 文本文件

观察并不总是那么容易。如果曾经使用过 Windows 或 Linux(在此之前，DOS)

一段时间，你可能会根据观察文件的方式感知到它们之间的差异。简单的文本文件能够在一个简单的文本编辑器中打开和检查。文字处理器文件能够在创建它的某种字处理软件里中打开。PowerPoint 演示文稿能够在 PowerPoint 应用程序里打开。如果尝试用 Word 或者 Excel 加载它，应用程序将显示乱码，或者(更可能)礼貌地拒绝服从打开命令。尝试用文字处理器或其他文本编辑器打开一个可执行程序文件，通常什么都得不到，或者得到屏幕上的一堆乱码。

文本文件(text file)是指可以在文本编辑器(如 Windows 中的记事本，或 Linux 下可用的诸多文本编辑器中的任何一款)中打开并能进行有意义地检查的文件。二进制文件(binary file)是指包含不能显示为有意义的文本的值的文件。大多数高端文字处理器通过操纵文本，然后再将其与(并不翻译成文本，而是用于指示一些诸如段间距、行高之类的事情的)格式信息组合在一起来混淆这一主题。在一个简单的文本编辑器中打开一个 Word 或 Office 文档，你就会明白我的意思。

文本文件包含大写字母、小写字母、数值数字，还有诸如标点符号之类的奇怪符号。有 94 个这样的可视化字符。文本文件还包含许多名为空白分隔符的字符。空白分隔符字符通过把文本文件拆分成行，并且在行内提供空格来使文件具有一定的结构。其中包括我们熟悉的空格字符、制表符字符、指示一行末尾的换行符字符，有时还有一些其他的字符。也有一些“化石”字符，如 BEL 字符，它在几十年前用于使电传打字机上的小机械铜铃响铃。虽然 BEL 从技术上来讲应该被认为是空白标识符，但是大多数简单的文本编辑器只是简单地忽略它。

在 PC 世界里，文本文件有些复杂，因为有另外 128 个字符包含数学符号的图像字符、有重音符号的字符和其他修饰符的字符、希腊字母字符，还有“方块”字符，它在图形用户界面(如 Windows 和 Gnome)问世之前，被广泛用于绘制屏幕格式。这些额外的字符在文本编辑器或终端窗口中的显示效果如何，完全取决于文本编辑器或终端窗口及其具体配置。

当通过 Unicode 标准引入非西方字母时，文本文件变得更加复杂。详细解释 Unicode 已经超出了本书的范畴，但是你可以在维基百科或其他地方找到关于它的详细介绍。

文本文件很容易显示、编辑和理解。唉，但是编程世界比文本文件包含更多的东西，在前面的章节中，我从计算机的角度定义了什么是计算机程序。打个比方，程序就是一个由非常小的步骤组成的漫长旅途。这些步骤是一个代表机器指令的二进制值的列表，而这些机器指令则指导着 CPU 做那些要完成手头工作所必需做的事情。这些机器指令即使是用它们的十六进制简写形式也相当令人类费

解。以下是以 16 进制表示的一小段二进制值序列：

```
    FE FF A2 37 4C 0A 29 00 91 CB 60 61 E8 E3 20 00 A8 00 B8 29 1F FF
69 55
```

它到底是不是一个真正的程序中的一部分呢？这可能必须要让 CPU 去找出答案了，除非你是一个自 1978 年以来从未见过的那种“机器代码疯子”。(它不是。)

但是，对于以这形式提交的程序，CPU 并没有感到任何麻烦。事实上，CPU 无法以任何其他方式处理程序。CPU 本身根本不能理解和服从如下所示的字符串：

```
LET X = 42
```

甚至这里称之为汇编语言的某些东西：

```
mov eax,42
```

对于 CPU 而言，它只是二进制。CPU 只可能将一个文本序列解释为二进制机器指令，但如果发生这种情况，那也是纯属巧合，而这种巧合不会超过三个或四个字符。这个指令序列也不可能做任何有用的事情。

从某一高度来看，汇编语言(或许多其他语言)编程的过程包括采用人类可读的文本文件，以某种方式将它们转化为含有 CPU 能够理解的二进制机器指令序列。作为程序员的你，则需要了解哪些文件代表哪些机器指令(详见后文)，以及每个机器指令是如何被处理的。此外，你还需要能够“打开”一个可执行的二进制文件并检查它所包含的二进制值。

5.1.2 用 Bless 编辑器查看文件内容

很幸运，有一些工具能够让你打开、显示和修改任意类型文件内的字符或者二进制字节。这些工具称为二进制编辑器(binary editor)或十六进制编辑器(hexadecimal editor)，在我的经验中，它们之中最好的(至少在 Linux 世界中)是 Bless 十六进制编辑器。它被设计为在图形用户界面(如 Gnome)下使用，并很容易通过浏览菜单找到。

Bless 在 Ubuntu 下不是默认安装的。可以从它的网页上免费下载：

http://home.gna.org/bless/

但是，你可以很轻松地从 Ubuntu 的 Application 菜单安装它。选择

add/remove 选项，并将该视图设置为 All(它是一项默认设置)。在 Search 关键字域中敲入 Bless，Bless 十六进制编辑器应是唯一出现的项。(给它几秒钟搜索时间；因为结果项不会瞬间出现。) 使它前面的复选框处于选中状态，以选择安装它，然后单击 Apply 按钮。一旦安装成功后，就可以通过 Application→Programming 菜单项使用 Bless 十六进制编辑器了，或者如果喜欢的话，可以为它创建一个桌面快捷方式。

对于 Bless 的演示，同时也演示了为什么对程序员而言，理解文本文件(甚至在字节层面上)非常有必要。在本书的清单存档(详见网址中的介绍)中有两个文件即 samwindows.txt 和 samlinux.txt。将它们都提取出来。启动 Bless，并使用 File→Open 命令，打开 samlinux.txt。当该文件已经打开后，再次使用文件→打开，打开 samwindows.txt。你将看到如图 5-1 所示的内容。

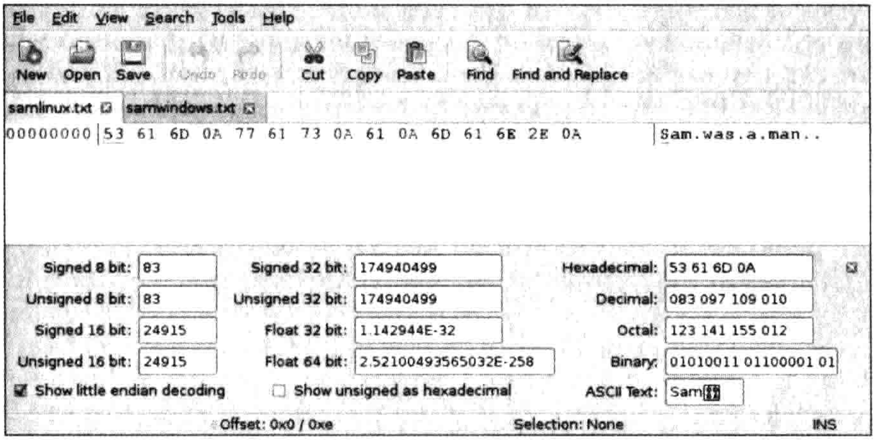


图 5-1 使用 Bless 十六进制编辑器显示一个 Linux 文本文件

这里，为了节省页面空间，我已经缩短了垂直方向的显示窗格，毕竟，文件本身仅有约 15 个字节长。每一个打开的文件在显示窗格上有都有一个选项卡，可以通过点击选项卡标签来方便地在各个文件之间进行切换。

显示窗格分为三个部分。左边一栏是偏移列。它包含显示在中间一栏里的那一行的第一个字节与文件开头的第一个字节的偏移量。该偏移量是以十六进制给出的。如果在文件的最开始位置，偏移列的值将是 00000000。中间一栏是 16 进制显示列。它以十六进制形式显示一行来自文件的数据字节。显示多少个字节取决于如何设置 Bless 窗口的大小以及屏幕的分辨率。显示字节数量的最小值为 17(有些奇怪)。中间栏的显示内容始终是十六进制，每个字节与相邻字节用空格

隔开。最右一栏是用可视的文本字符以文本形式显示的同一行数据。那些不可显示的二进制值采用句号字符表示。

如果单击 samwindows.txt 选项卡，将会看到以相同方式显示的、使用 Windows 记事本文本编辑器创建的另一个文件。samwindows.txt 文件有点长，中间一栏将有第二行数据字节。第二行偏移量是 00000012。它是以十六进制表示的第二行首字节的偏移量。

为什么这两个文件会不同呢？弹出一个终端窗口，并使用 cat 命令来显示这两个文件。在两种情况下的显示结果相同：

```
Sam
was
a
man.
```

图 5-2 展示了显示 samwindows.txt 的 Bless 编辑器。当 Bless 显示这两个文件的时候，请仔细观察它们(或者观察图 5-1 和图 5-2)，并在继续阅读本书之前，尝试自己独立找出两个文件之间的差别。

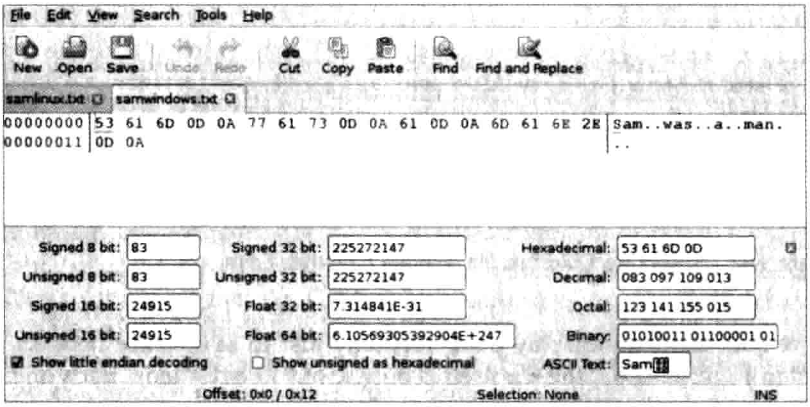


图 5-2　用 Bless 编辑器显示一个 Windows 文本文件

在这两个文件中，每行文本末尾都是一个 0AH 字节。该文件的 Windows 版本还有一点额外的内容：在每一个 0AH 字节前面都有一个 0DH 字节。Linux 文件没有 0DH 字节。尽管已经标准化为“纯文本文件”，根据创建文件的操作系统的不同，仍然可能会有一些小的差异。根据约定，Windows 文本文件(和以前的 DOS 文本文件)每行的末尾使用两个字符进行标识：0DH，后面跟 0AH。而 Linux(和几乎所有的 Unix 操作系统的后裔)只用一个 0AH 字节标识每行的结尾。

正如使用 cat 看到的两个文件一样，Linux 能以相同的结果精确地显示这两个版本。然而，如果你打算采用 Linux 版本，并使用 Windows 记事本文本编辑器加载该文件，将会看到有些不同，如图 5-3 所示。

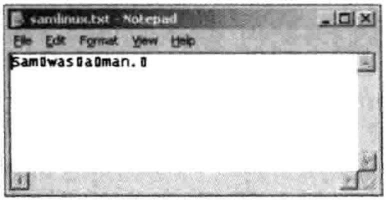


图 5-3 在 Windows 下显示一个 Linux 文本文件

记事本期望在每个文本行的末尾都看到 0DH 和 0AH，它不理解只有一个 0AH 孤零零地作为行结束(EOL)标记。相反，无论在何处看到 0AH，它都会插入一个窄矩形块，因为在不知道如何显示或解释任何单个字符时，它就会这样做。并非所有 Windows 软件都这么挑剔。许多或者说大多数其他 Windows 实用工具都了解 0AH 是一个非常好的 EOL(行结束)标记。

在每行末尾的 0DH 字节是又一个“化石”字符的例子。几十年前，在电传打印机时代，有两个单独的电命令被内置到电传机器中，当打印文档时，用它们来处理文本行的结束。一个命令用于索引文件的下一行，另外一个命令用于返回到左边距处的打印开始处。它们分别被称为换行和回车。回车被编码为 0DH，换行被编码为 0AH。大多数计算机系统和软件现在忽略回车码，但是也有少数(如记事本)仍然需要它来正确显示文本文件。

对于你来讲，这种文本文件标准之间的小差异不会带来大的问题，而且如果把文件从 Windows 下导入 Linux 中，你可以轻松地以手动方式删除额外的回车符，或者用汇编语言编写一个小程序来替你完成。现在，对于你来讲，重要的是了解如何加载一个文件到 Bless 十六进制编辑器(或者其他任何你喜欢的十六进制编辑器，有很多种)中，并在单个字节级别上检查该文件。

除了查看之外，Bless 还能做更多的事情。编辑一个已加载文件既可以通过中间(二进制)栏也可以通过最右(文本)栏来实现。可以通过 Tab 键在这两栏之间移动编辑光标。无论在哪一栏，都可以使用单独的箭头键，一个字节一个字节地移动光标。Bless 识别 Insert 键状态，你可以直接敲入字节或者适当地插入字节。

一旦已经对某一文件做出修改，要单击“保存”按钮将其存回磁盘。

5.1.3 解释原始数据

将文本文件看作由十六进制值组成的行是计算的基本准则：一切都是由位构成的，位模式意味着我们认同的它们的含义。Bless 中显示的两个文本文件都以大写字母“S”开头，它以十六进制表示是 53H，53H 也可以被认为是十进制数 83。在最底层，它是一个 8 位的模式：01010011。在这个文件中，我们约定使用位模式 01010011 代表大写字母“S”。在一个二进制可执行文件中，位模式 01010011 可能意味着完全不同的内容，这取决于它刚好位于文件中的位置，以及在该文件中，它的附近存在着哪些其他位模式。

这就是 Bless 十六进制编辑器的下部为什么存在一个窗格的原因。它采用始于光标处的字节序列，并且显示这些字节的所有不同解释方法。请记住，你不会总是使用像 Bless 一样的十六进制编辑器来查看文本文件。你可能会检查自己编写的程序所产生的数据文件，而那个数据文件可能代表 32 位有符号整数序列；或者无符号 16 位整数序列；或者 64 位浮点数字序列；或任何以上所有类型的组合。在中间窗格中看到的所有内容都是一系列十六进制值。这些值代表什么，取决于什么程序把这些值写入该文件，以及这些值在现实世界中所代表的含义。它们是美元金额？测量值？某种类型的仪器所产生的数据点？这取决于你和你所使用的软件。像所有文件一样，该文件只不过是一个存储在某处的二进制模式序列，我们(使用 Bless)将其以十六进制进行展示，以便更加容易理解和修改。

在中间栏的十六进制值列表中移动光标，并观察底部窗格中列出的解释如何变化。请注意，有些解释看起来只有一个字节(8 位)，而有些解释则有两个字节(16 位)，或者 4 个字节(32 位)，或者 8 个字节(64 位)。在任何情况下，被解释的字节序列都是从光标所在位置开始，然后向右侧扩展。例如，对于文件中第一个位置的光标：

- 53H 可能被解释为十进制值 83。
- 53 61H 可能被解释为十进制值 21345。
- 53 61 6D 0AH 可能被解释为十进制值 1398893834。
- 53 61 6D 0A 77 61 73 0AH 可能被解释为浮点数 4.54365038640977^{93}。

(有符号值和无符号值之间的差异要等到本书后面部分再介绍。)重要的是要明白，在所有情况下，它们都是同一文件中的同一个位置处的同样字节序列。所不同的是我们查看多少个字节以及我们希望这些字节序列代表什么类型的值。

当我们开始编写关于数字的程序后，这一点可能会更加清楚。

5.1.4 “字节序”

在 Bless 编辑器底部窗格的左下角，有一个标记为“show little endian coding”(显示小端法解码)的复选框。默认情况下，该复选框处于未选中状态，但是在几乎所有情况下，它应该处于选中状态。这个复选框告诉 Bless 是按照“大端法(big endian)”顺序还是按照“小端法(little endian)”顺序将字节序列解释为数字。如果选中或者取消选中该复选框，显示在窗格中的值将发生根本性的改变，即使不移动光标。当改变该复选框的状态时，正在改变的是编辑器将文件中的一些字节序列解释为某类的数字所采用的方法。

如果回想一下第 4 章，你可能还记得，一个字节能够代表从 0 到 255 之间的任一数字。如果想表示一个大于 255 的数值，必须使用多个字节来实现。一行中的两个字节序列可以代表从 0 到 65 535 之间的任何数字。然而，一旦使用不止一个字节来代表一个数字值，字节的顺序就变得至关重要。

让我们回到前面 Bless 中加载的两个文件中任意一个的最前面两个字节。它们名义上是字母“S”和“a”，但实际上只不过是另一种解释而已。十六进制序列 53 61H 也可以解释为数字。53H 先出现在文件中。61H 后出现在文件中(见图 5-1 和图 5-2)。因此，合在一起作为一个单个的 16 位值，这两个字节变成了十六进制数 53 61H。

或者它们还能变成别的什么吗？也许你觉得这个问题有点古怪，但它并不那么简单。参见图 5-4，该图的左边部分是前述文本文件示例在 Bless 的十六进制显示窗格中的一小部分内容。它只显示前两个字节以及它们相对于文件开始位置的偏移量。该图的右边部分是同样的信息，但是左右反向，好像在镜子里看到的一样。这是相同顺序的相同字节，但是我们以不同的方式来看待它们。我们先前假定的是 16 位十六进制数 53 61H，现在看起来成了 61 53H。

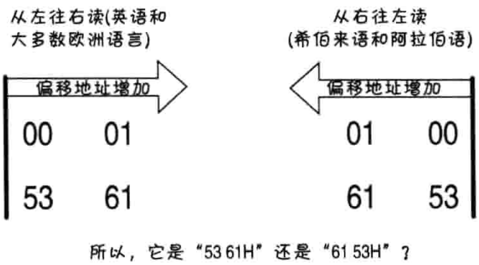


图 5-4 显示顺序不同 vs.计算顺序不同

难道是数字变了？至少不是从计算机的角度。所有这些改变只是我们把它们印在本书书页上的方式。按照习俗，人们从左往右阅读英文。Bless 十六进制编辑器的显示结果反映了这一点。但是，在世界上还有许多种其他语言，包括希伯来语和阿拉伯语，它们是从右向左阅读的。阿拉伯语程序员的第一冲动可能是把那两个字节看成 6153H，特别是如果他或她正在使用为(从右往左显示文件内容的)阿拉伯语习惯而设计的软件时。

实际情况比上述情况更加混乱。西方语言(包括英语)有点“精神分裂”，因为它们从左向右阅读文字，但是从右往左计算数字列。数字 426 由 4 个 100、2 个 10 和 6 个 1 组成，而不是 4 个 1、2 个 10 和 6 个 100 组成。这里是按照西方习惯，最不重要的列在最右侧，然后从右往左列值增加。最重要的列在最左边。

在计算中，混乱绝对不是什么好事情。因此，在一个多字节数字中，是从左到右还是从右到左显示字节序列，必须达成共识：到底哪一个字节代表最不重要的数字，哪一个字节代表最重要的数字。在计算机中，我们有两个选择：

- 我们可以约定，多字节值的最低有效字节(the least significant byte)的偏移量最小，最高有效字节(the most significant byte)的偏移量最高。
- 我们也可以约定，多字节值的最高有效字节的偏移量最小，最低有效字节的偏移量最大。

这两个选择是相互排斥的。计算机操作必须二者选其一，不能在任一程序中同时使用它们。此外，这个选择并不限于操作系统或者某个特定程序。它被烧制到 CPU 芯片及其指令集上。将多字节值的最低有效字节存储到最低偏移位置的计算机体系结构称为*小端系统*(*little endian*)。将多字节值的最高有效字节存储到最低偏移位置的计算机体系结构称为*大端系统*(*big endian*)。

通过图 5-5 应该看得更清楚些。在大端法中，多字节值从它的最高有效字节开始。在小端法中，多字节值从它的最低效字节开始。可以这样认为：大端法，大的先结束；小端法，小的先结束。

这里有非常重大的差别！在上述示例中，文本文件的开始两个字节在大端法里表示十进制数 21 345，但是在小端法里却表示十进制数 24 915。

在根本不知道系统“字节序”的情况下编写相当多的程序是可能的。如果使用高级语言(如 Visual Basic、Delphi 或者 C)，大多数字节序的影响被语言和语言编译器所隐藏，至少在发生底层错误之前。一旦开始在字节级别上读取文件，必须要知道如何阅读它们；如果使用汇编语言编程，最好能对字节序的介入应对自如。

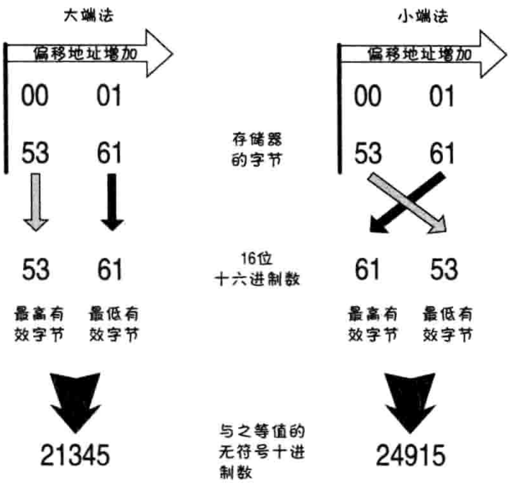


图 5-5　16 位值的大端法 vs.小端法

阅读大端系统中的数字数据的十六进制显示很容易，因为它们是按照西方人期待的顺序出现的，最重要的数在最左边。在小端系统中，一切都是颠倒的；表示一个数字使用的字节越多，它就变得越混乱。图 5-6 显示了采用不同字节序计算一个 32 位值的区别。小端法程序员像阅读希伯来文或阿拉伯文一样从右到左阅读多字节值的十六进制显示。

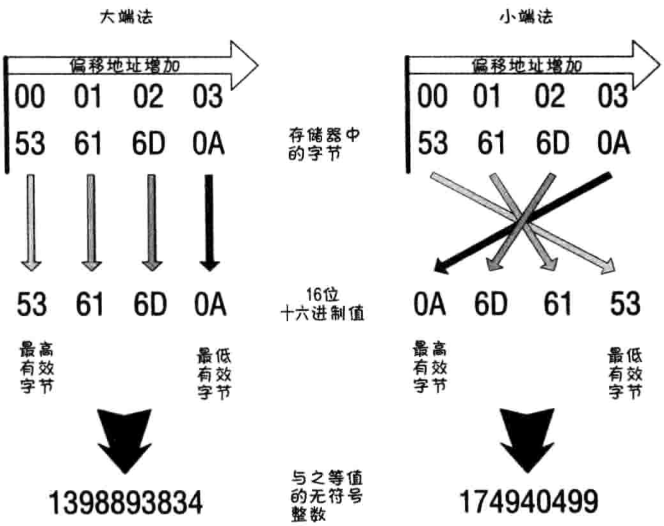


图 5-6　32 位值的大端法 vs.小端法

请记住，字节排列顺序的差异不仅适用于存储在文件中的字节，而且也适用于存储在内存中的字节。当使用调试器(我将稍后解释)查看存放在内存中的数值时，所有的规则同样适用。

那么，Linux 系统使用哪种字节序呢？两个都用！(但是不能同时使用。) 同样，这也不关操作系统的事。整个 x86 硬件体系结构，从低端的 8086 一直到最新的 Core 2 Quad，采用的都是小端字节序。其他的硬件体系结构，如摩托罗拉的 68000 和早期的 PowerPC，以及 IBM 大型机架构(如 System/370)，采用的都是大端字节序。最近的硬件体系结构都被设计为双向字节序，这意味着它们可以在硬件级别上配置(虽然有一些困难)为以一种方式或者以另外一种方式解释数字。Alpha、MIPS 和 Intel 公司的安腾体系结构都是双向字节序。

如果(最有可能)正在一个普通的 x86 处理器上运行 Linux，那么你采用的是小端字节序，应该让 Bless 的标识为“显示小端法解码”的复选框处于选中状态。其他编程工具可能会提供一些有关大端法显示或小端法显示的选项。要确保无论使用任何工具，都让正确的选项处于选中状态。

当然，Linux 操作系统能够在任何硬件架构上运行，因此使用 Linux 并不能保证你将面对一个大端系统或者小端系统，这也是我为什么在这里详尽讲解字节序相关知识的原因。可以通过研究系统来了解当前生效的字节序，也可以通过如下方式的检查来了解它：将一个 32 位整数存入内存，然后使用调试器或者诸如 Bless 之类的十六进制编辑器来对其进行查看。如果了解十六进制(最好这样！)，系统字节序将一目了然。

5.2　文本进去，代码出来

从某一高度来看，所有程序都归结为文件处理。其目标是采用一个或多个人类可读的文本文件，然后对其进行处理，创建一个可执行程序文件，你可以在任何正在使用的操作系统下加载和运行它。对于本书而言，操作系统将是 Linux，但是我在本节介绍的一般过程适用于几乎任何种类的编程和几乎所有的操作系统。

编程是一个过程，它会由于语言及其支持工具的不同而有很大的不同。在现代图形化交互开发环境中，如 Visual Basic 和 Delphi，许多文件处理在“幕后”完成，而作为程序员的你，则只是盯着一个或多个显示在面前的文件，思考下一步行动。在汇编语言中，情况并非如此。大多数汇编语言程序员可以使用更简单

的工具，通过一个命令行或一个脚本文件，明确地将这些文件处理成离散的步骤。

然而，这些步骤完成之后，将文本文件转换为二进制文件的一般过程是一种翻译过程，完成这项工作的程序叫作翻译器(translator)。翻译器是一种接受人类可读的源文件并将其生成某种二进制类型文件的程序。输出的二进制文件可能是 CPU 可以理解的可执行程序文件，也可能是字体文件(font file)，或者压缩二进制数据文件，或者一百种二进制文件类型中的任何其他类型。

程序翻译器(program translator)是生成 CPU 可以理解的机器指令的翻译程序。程序翻译器一行一行地读入源代码文件，并写出一个完成源代码文件所描述的计算机操作的二进制机器指令文件。这种二进制文件称为目标代码文件(*object code file*)。

编译器(compiler)是一种程序翻译器，它读入采用高级语言(如 C 或 Pascal)编写的源代码文件，并写出目标代码文件。

汇编编译器(assembler)是一种专用编译器。它也是一种程序翻译器，用于读取源代码文件并将其输出为供 CPU 执行的目标代码文件。但是，汇编编译器专门设计用来把汇编语言翻译成目标代码。语言编译器(如 Pascal 或 C++编译器)将源代码文件编译为目标代码文件，采用类似的描述，我们说，汇编语言编译器将汇编语言源代码文件编译为目标代码文件。翻译过程在这两种情况下是类似的。然而，汇编语言具有非常重要的特征，也正是这一点，使它在众多的编译器中与众不同，即：完全控制目标代码。

5.2.1　汇编语言

有些人把汇编语言定义为一种语言，在这种语言中，一行源代码生成一条机器指令。但绝不能按照字面意义去理解，因为汇编语言源代码文件中的某些行是翻译器程序(而不是 CPU)的指令，根本不产生机器指令。

这里是一个比较好的定义：

汇编语言(assembly language)是一种翻译器语言，它允许完全控制由翻译器程序生成的每一条机器指令。这种翻译器程序被称为汇编编译器。

恰恰相反，Pascal 或 C++编译器对于如何将给定语言的语句翻译成机器指令序列做出了许多不可视的、不容更改的决定。例如，下面的 Pascal 语句将一个名为 I 的数字变量赋值为 42：

```
I := 42;
```

当 Pascal 编译器读取此行时，它将输出四五条机器指令，这些指令获取字面数字值 42，将其存储在名称 I 编码的内存位置。通常情况下，Pascal 程序员根本不知道那四五条指令实际上是什么，也完全没有办法改变它们，即使知道一个比编译器使用的机器指令序列更加快速、更加高效的指令序列。Pascal 编译器有自己产生机器指令的方式，除了被动接受它写入目标代码文件中的所有内容，以完成写入源代码文件的 Pascal 语句的工作之外，你别无选择。

为了公平起见，现代的高级语言编译器通常采用所谓的行内汇编(in-line assembly)，它允许程序员从编译器“收回”控制权，“插入”一系列他或她自己设计的机器指令。很多现代汇编语言采用这种方式工作，实际上它被认为是一种先进的技术，因为在能够使用行内汇编以期“做得更好”之前，你必须首先了解编译器如何生成自己的代码(并且在没有进行大量的学习研究和实践之前，不要像很多人那样，想当然地认为自己能够比编译器做得更好)！

汇编编译器为它产生的每一条机器指令至少要查看一行源代码文件。它通常情况下要查看若干行源代码，其他那些行处理其他各种事情，但在最终的目标代码文件中，每一条机器指令都由源代码文件中的一个相应行控制。

CPU 有很多机器指令，其中每一条都有一个对应的汇编语言助记符。顾名思义，这些助记符的目的就是为了帮助程序员记住二进制机器指令。例如，能够实现将标志寄存器压入堆栈的二进制机器指令是 9CH，它的助记符是 PUSHF，这比记忆 9CH 容易多了。

编写汇编语言源代码文件实际上是在组织一系列助记符，通常情况下，在源代码文本文件中每行放一个助记符。一个源代码文件中的部分内容可能是这样的：

```
mov eax,4                ; 04H 指定系统写内核调用
mov ebx,1                ; 01H 指定标准输出
mov ecx,Message          ; 把要显示的字符串的起始地址加载到 ECX 中
mov edx,MessageLength    ; 把要显示的字符串的字符个数加载到 EDX 中
int 80H;                 ; 进行内核调用
```

这里，左边距处的 MOV 是助记符。紧挨着每一个助记符的数字和文本叫作操作数。各种各样的机器指令拥有各种各样的操作数，也有一些指令(如前面所述的 PUSHF)根本不使用操作数。

助记符和它的操作数合在一起称为指令(instruction)。(分号右侧的语句是注释，它们不是指令的组成部分。)本书中大部分内容将使用这种注释语句来对 CPU 的纯二进制机器指令的“人类可读代理”进行说明。如果特别谈论二进制代码的话，通常指的是机器指令。

汇编编译器最重要的工作是从源代码文件中读出一行行语句并将机器指令写出到目标代码文件(见图 5-7)。

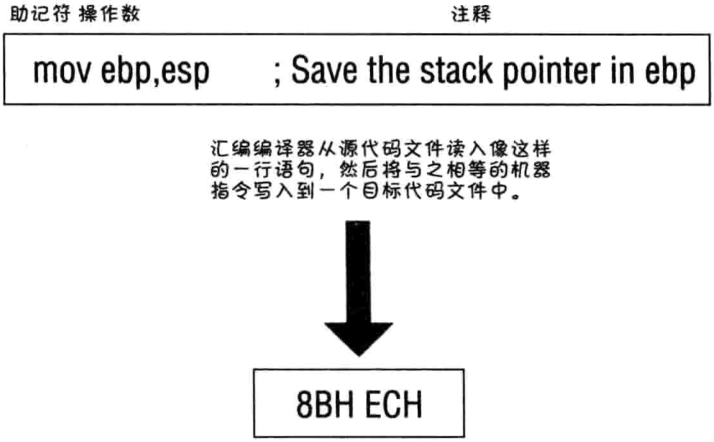


图 5-7　汇编编译器的工作

5.2.2　注释

每条指令的右侧是以分号开始的文本。该文本叫作注释，其目的显而易见：解释相关汇编语言指令的作用。指令 MOV EBX, ESP 把当前的堆栈指针值放入寄存器 ebx，但是为什么要这样做呢？在你编写的汇编语言程序的上下文中，该条指令能够完成什么任务呢？注释给出了原因。

在结构上，注释始于一行的第一个分号，一直持续到标识该行结尾的 EOL。注释并不需要与指令位于同一行。在汇编语言程序的注释块中有许多有用的描述，这些注释块儿由只包含纯注释的一系列文本行组成。其中每一行都以左边距处的分号为开始。

注释对于汇编语言程序的成功而言至关重要，其重要程度远远超过在任何一种其他编程语言。我的个人建议是源代码文件中的每一条指令都应该有关于其功能及作用的相应注释。此外，在每组以某种方式共同工作的指令前面，都应该有一个注释块，从某一高度解释一下这组指令，以及它们如何协同工作。

注释对于理解文本文件的结构非常重要，因为在汇编语言中，注释结束于每一行的结尾。在大多数其他语言中(如 Pascal 和 C)，注释都放在注释分隔符(如*和*)之间，并且每行结尾的 EOL 标记被忽略。

简而言之：注释开始于分号，结束于 EOL。

5.2.3　当心“只写”源代码

汇编语言中有一种非常严重的问题，我觉得现在是指出来的最佳时机。这些指令本身非常简洁，以至于任何有用的事情都需要许多条指令才能完成。除此之外，尽管每条指令声明了它所做的事情，但是并没有任何内容表明该条指令操作的上下文环境。在 Pascal 或者 Basic 代码中，可以通过一些技巧和规则(以及一些见名知义的标识符)构建上下文环境，但是在汇编语言中，只能通过注释来添加上下文环境。

如果没有上下文环境，汇编语言就变成了所谓的“只写”代码。情况可能会这样：11 月 1 日，在创作灵感高涨时，你完成了一个能够做某些重要事情的小实用工具程序中的约 300 条指令。1 月 1 日你又回到该程序，打算为它添加一个新特性，但是却发现，已经不再记得它的工作原理了。单个指令都是正确的，程序汇编和运行都符合预期，而从某一高度了解这一切如何组织到一起以及如何工作，却消失在圣诞节的回忆和 8 个星期埋头于其他事情的重压之下。换句话说，你写了这个程序，但是却不能阅读它，或者修改它了。瞧！这就是“只写”代码。

尽管注释的确占用源代码磁盘文件的空间，但是它们不会被复制到目标代码文件中，而且一个程序在源代码中添加注释和根本不添加任何注释的情况下运行速度是一样的。编写汇编语言程序需要投入大量的时间和精力——远远超过在那些“一半天堂”的语言(如 C 和 C++)，并且匪夷所思地超过那些“什么都替你做了”的 IDE 集成环境(如 Delphi、Lazarus 和 Gambas)。汇编语言程序的编写方式比任何一种其他语言程序的编写方式都困难；如果不添加注释，你很可能最终只能丢弃令人费解的数百行代码，从头开始，再编写一遍。

聪明地工作。不停地写注释，直到退出。

5.2.4　目标代码和连接器

汇编编译器读入源代码文件，并生成包含 CPU 能够理解的机器指令以及所有在源代码中定义的数据的目标代码文件。

读入源代码并写出已完成的可执行程序作为其目标代码文件，这似乎是汇编编译器天经地义的责任，但是这种(像上一自然段中所描述的)预期目标几乎很少

完成过。我在本书中所讲的汇编编译器(NASM)能够为 DOS 程序做这件事情，并能够为实模式平面模型写出.com 可执行文件。更为现代的操作系统(如 Linux 和 Windows)对于做这件事情而言则太过复杂，说实话，这样的一步到位汇编并没有什么真正的价值，除非你刚刚开始学习汇编语言。

因此，现代汇编编译器产生的目标代码文件是一种源代码和可执行程序之间的中间步骤。这个中间步骤是一种类型的二进制文件，叫作目标模块(object module)，或者只是一个简单的目标代码文件。

目标代码文件本身并不能作为程序运行。要想把目标代码文件变成可执行程序文件，还需要一个额外的必要步骤，那就是连接(link)。

目标代码文件作为中间步骤的原因是：大的源代码文件可以划分成许多更小的源代码文件，以保持文件大小和复杂性的可管理性。汇编编译器单独汇编各种各样的片段，然后将生成的目标代码文件交织在一起，组成一个可执行程序文件。该过程如图 5-8 所示。

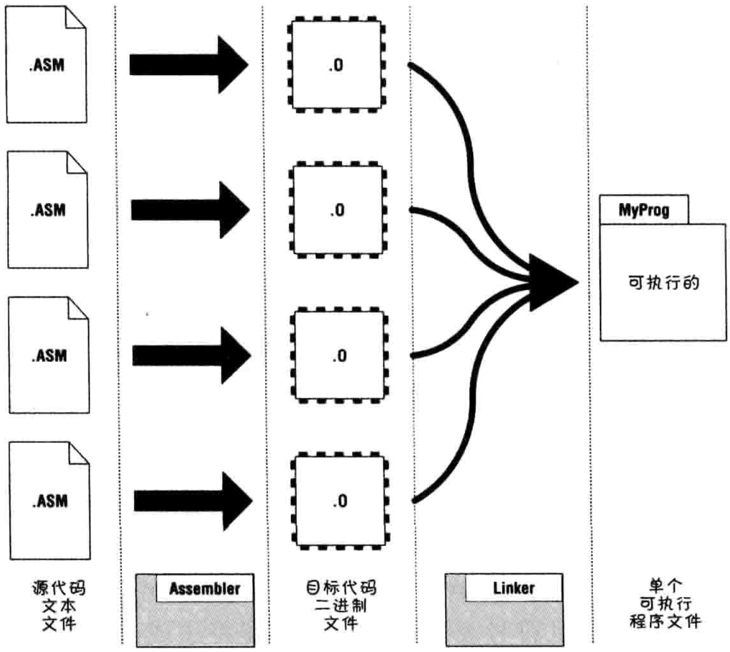


图 5-8　汇编编译器 assembler 和 连接器 linker

当第一次学习汇编编程时，不太可能编写分布在多个源代码文件中的程序。这可能使得连接器看起来无关紧要，因为只有一个程序块，没有什么需要连接

的。但是事实并非如此：连接器不只是把目标代码块组合成一个单个的块。它确保模块以外的函数调用能够到达目标对象模块，并且确保所有此类内存引用实际上指向它们期望引用的地方。汇编编译器的工作是显而易见的；连接器的工作是微妙的。两者对于产生一个已完成的、能够工作的可执行文件来说，都是必不可少的。

此外，你很快就要进入正题：开始将程序中经常使用的部分提取到自己的个人代码库。这样做有两个原因。

(1) 可以把已通过测试和验证的程序放到单独的库文件中，将其连接到任何可能需要它们的程序中。这样就可以重用代码，而不是每一次都重蹈覆辙，重新开始一个新的汇编语言编程项目。

(2) 一旦一个程序的部分内容已经通过测试，并且发现是正确的，就没有必要浪费时间让它们随较新的、未经测试的程序部分一起，一遍又一遍地重新编译。一旦一个主要程序增长到成千上万行代码(这一时刻可能会比你预想的更快到达！)，通过只编译当前正在工作的那部分程序，然后将已完成部分连接到最终程序，不必每次重新编译整个程序的每一个组成部分，可以节省大量的时间。

连接器的工作非常复杂，而且不容易描述。每个目标模块可能包含以下内容：

- 程序代码，包括已命名过程。
- 对外部模块中存放的已命名过程的引用。
- 已命名数据对象，如：具有预定义值的数字和字符串。
- 为程序以后使用预留的、只是空白空间的已命名数据对象。
- 对存放在模块外部的数据对象的引用。
- 调试信息。
- 一些用来帮助连接器创建可执行文件的其他不太常见的杂项。

要想将若干个目标模块处理成一个单个的可执行模块，连接器必须首先建立一个索引，这个索引叫作符号表(symbol table)，里面存放的是它连接的每一个目标模块中的每一个已命名项，其中存放着一些关于哪个名字(叫作符号)指向模块内部哪个位置的信息。一旦符号表完成后，连接器将生成一个映像，里面存放着当操作系统加载它时，如何将可执行程序安排在内存中的相关信息。然后，该映像被作为可执行文件写入磁盘。

关于映像，最重要的事情是：连接器生成相关的地址。一个目标模块允许引

用其他目标模块中的符号。在汇编过程中，这些外部引用(external reference)被作为“小洞”预留下来，以后再进行填补——非常顺理成章，因为这些外部符号存在的模块可能还没有编译，甚至还没有编写。当连接器生成一个最终可执行程序文件的映像时，它知道所有这些符号位于该映像的什么地方，从而可以把所有的实际地址放入那些外部引用“小洞”。

从某种意义上来讲，调试信息是倒退了一步：先前在编译过程中被剔除的那部分源代码又被汇编编译器放回目标模块中。源代码的这部分程序大多是数据项和过程名称，它们被嵌入目标文件中，因而当调试的时候，程序员(你！)更容易看到数据项的名字。(我将在以后对其进行更深入地讲解。)调试信息是可选的，也就是说，连接器并不需要它来建立一个正确的可执行文件。当你仍然工作于某个程序的时候，可以选择将调试信息嵌入到目标文件中。一旦程序制作完成，并完全通过调试，再一次在不带调试信息的情况下重新运行汇编编译器和连接器。这是很重要的，因为调试信息可以让可执行文件变得比不带调试信息大很多，而且通常情况下速度会慢一些。

5.2.5 重定位能力

最早期计算机系统(如运行 CP/M-80 的 8080 系统)的内存体系结构非常简单。程序也被设计为加载并运行在一个特定的物理内存地址。对于 CP/M 而言，这个地址是 100H。程序员可以认为，任何程序都将从 100H 开始，然后从那里往后延伸。数据项和过程的内存地址是实际的物理地址，程序每一次运行时，其数据项都精确地以相同的内存位置加载和引用。

随着 8086 及其特有的操作系统的到来(如 CP/M-86 和 PC DOS)，一切都发生了变化。与 8086 一同引入的 Intel 体系结构内部有了很大的改进，从而使程序无须被编译为在特定的物理地址运行。可执行文件内部的所有引用都通过相对于该程序起始位置的偏移量来指定，例如，变量将不再被可靠地定位于物理地址为 02C7H 的内存中。相反，它被可靠地定位于相对文件起始位置的某个偏移位置处。偏移量总是相同的，并且因为所有的引用都是相对于可执行程序文件的起始位置，所以，在运行时，程序放在物理内存中的什么地方都没有关系。

这一特性叫可重定位能力，它是所有现代化计算机系统的必要组成部分，特别是当多个程序可以同时运行时。处理重定位也许是连接器的最主要工作。幸运的是，它自己能够完成此项工作，并不需要任何来自外界的输入。一旦任务完成且善后工作处理完毕，文件翻译工作就彻底完成，你将得到一个可执行程序

文件。

5.3 汇编语言开发过程

正如你所看到的那样，在编写、编译、测试汇编语言程序的过程中，包含很多种不同类型的文件和相当多的程序。这个过程本身比听起来更加复杂。我已经画出了一个“地图”，以帮助你在本章其余的讨论中仍然保持“风度”。图 5-9 展示了从某一高度来看，汇编语言开发过程采用的最常见形式。乍看之下，它可能像一幅洛杉矶高速公路系统中的地图，但实际上该流程相当简单，在花了几个晚上搞定某个程序之后，你会对这个流程非常熟悉，甚至于它将成你的潜意识行为。

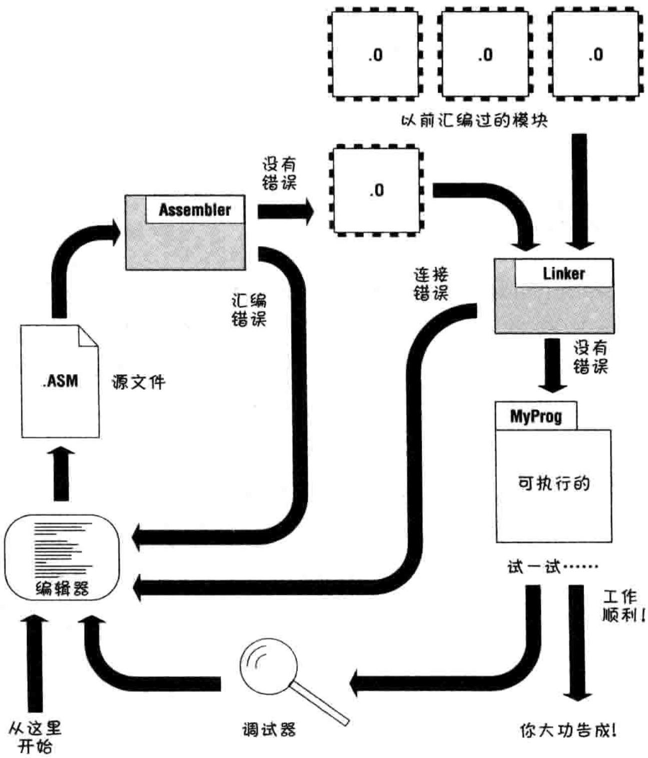


图 5-9　汇编语言的开发过程

概括地讲，这个过程可归结为如下内容：

(1) 在文本编辑器中创建汇编语言源代码文件。

(2) 用汇编编译器从源代码文件创建目标模块。
(3) 使用连接器把目标模块(和任何以前编译的、也是该项目的一部分的目标模块)转换为单个的可执行文件。
(4) 通过运行程序文件来对其进行测试，如果有必要，可以使用调试器。
(5) 回到步骤(1)中的文本编辑器，修复以前可能犯下的任何错误，并在必要时编写新的代码。
(6) 重复步骤(1)到步骤(5)，直到完成。

5.3.1 工作目录规范

程序员通常从 0 开始计数，如果要对汇编语言开发过程的步骤进行计数的话，那么，步骤 0 包含的内容为：在 Linux 电脑上建立一个系统目录，以管理在该过程中创建和处理的文件。

这里有一个规则，你需要了解并且坚决采纳：一个项目只能存储于一个目录之下。也就是说，当你想编写一个名为 TextCaser 的 Linux 程序时，创建一个名为 TextCaser 的目录，并保持在该目录中只保存与 TextCaser 项目直接相关的文件。如果你有另一个名为 TabExploder 的项目，应该为其单独创建一个自己的目录。首先，这是一种非常好的管理实践，其次，它能有效防止将生成文件(makefile)搞乱。(在后面部分，当我们开始着手创建文件和生成文件时，我会对其进行更加详细的介绍。)

我建议为汇编开发项目建立一个目录架构，我的经验建议这样做：在 Linux Home 目录下创建一个名为“asmwork”的目录(或采用另外一些适当的描述性名称)，并创建一个以个人项目命名的目录作为整个汇编语言目录的子目录。

顺便说一下，一个目录的名称可以与该项目的主.ASM 文件相同；也就是说，textcaser.asm 非常乐意住在名为 textcaser 的目录下。

此时，如果你还没有下载并解压缩本书的存档文件列表，我建议你先做那件事——在本节，你将会用到其中一个范例存档文件。该存档文件叫 asmsbs3e.zip，它可在以下网址中找到：

www.copperwood.com/pub

或者，采用另外一个备用网址：

www.junkbox.com/pub

(我在两个不同的互联网主机服务上有两个域名，以便于至少有一个总是处于工作状态，并且可用。文件是相同的，无论从哪个网站上都能够下载。)

当解压时，存档文件列表将在你选择的任意一个父级目录上创建单独的项目目录。我建议你把它解压到 asmwork 目录，或者任何你最终命名的目录。

5.3.2 编辑源代码文件

实际的开发过程从输入程序代码到文本编辑器中开始。使用哪个文本编辑器并不重要，你可以从几十个中进行选择。(我将在下一章中推荐一个非常好的文本编辑器。)

唯一重要的事情是要记住，诸如 Microsoft Word 和 Open Office Writer 之类的文字处理器会在文档文件中嵌入大量额外的二进制数据，它们远远多于敲入的文本。这些二进制数据控制着很多东西，如行间距、字体和字号、页眉和页脚以及许多汇编编译器不需要，也不知道的其他事情。汇编编译器并不总是擅长忽略此类数据，这可能会在编译时导致一些错误。

Ubuntu Linux 操作系统预安装的文本编辑器——gedit(可通过 Application→Accessories 菜单找到)是非常好的一款，特别是当你刚刚开始学习汇编编程，不需为单个程序处理成千上万行代码的时候。

嗯，至于你如何想出输入什么内容，这是另一个单独的问题，我将在第 8 章的大部分篇幅进行阐述。你当然会有一堆的注释，也许是一些伪代码，一些图表，也许是一幅正式的流程图。这些都可以通过工具软件在屏幕上完成，或者用铅笔在方格信笺纸上完成。(也许我有点老土，但我仍然使用信笺和铅笔。)

汇编语言源代码文件几乎都以.ASM 为文件扩展名保存到磁盘上。换言之，对于一个名为 MyProg 的程序，其汇编语言源代码文件将被命名为 MyProg.asm。

5.3.3 编译源代码文件

正如你在图 5-9 中可以看到流程一样，文本编辑器生成一个以. ASM 为扩展名的源代码文本文件。然后这个文件被传递给汇编编译器程序本身，以将其解释为一个目标模块文件。在本书中我所侧重的环境——Linux 和在 NASM 汇编编译器下，文件的扩展名将是.O。

通过命令行调用汇编编译器时，需要向其提供一个打算让它处理的源代码文件的名称。Linux 将从磁盘加载汇编编译器并运行它，然后汇编编译器将打开命令行中指定名称的源代码文件。转瞬之间(特别是你将在本书中找到的较小的学习程序)，它将创建一个与源文件具有相同名称的目标文件，但是以.O 为文件扩

展名。

当读取源代码文件中的代码行时，汇编编译器会对它们加以检查，建立一个符号表来总结源代码文件中的每一个已命名项，构建源代码行所代表的二进制机器指令，然后把这些机器指令和符号信息写入目标模块文件。当汇编编译器完成并关闭该目标模块文件时，它的工作完成，它将终止运行。在现代的 PC 上，对于一个不到 500 行代码的程序而言，这些事情的发生只需 1 秒或者更少的时间。

5.3.4 汇编错误

注意，前面的段落描述了如果.ASM 文件是正确的，将发生什么情况。提到正确，我的意思是该文件是完全为汇编编译器所理解的，也能够被翻译成机器指令，没有使汇编编译器迷惑不解。如果从源代码文件中读取一行代码时，汇编编译器遇到一些不能理解的事情，这个不被理解的文本叫错误(error)，汇编编译器会显示一个错误信息(error message)。

例如，下面的汇编语言行会使汇编编译器迷惑不解，它会提示一个错误信息：

```
mov eax,evx
```

原因很简单：没有 EVX 这个东西。代码中的 EVX 实际上本来应该是“EBX”，这是一个 CPU 寄存器的名字。(在键盘上，V 键旁边的是 B 键，在手指不一定知道情况下，你可能误敲了 V 键，于是错误就这样产生了！)这样的错别字是迄今为止最简单的一种位置错误。其他一些错误则需要一定的研究才能发现，而且涉及汇编编译器的许多规则——在大多数情况下，它们同样也是 CPU 的规则。例如：

```
mov ax,ebx
```

这条代码看起来似乎是正确的，因为 AX 与 EBX 都是真正的寄存器。然而，又一想，你可能会注意到，AX 是一个 16 位寄存器，而 EBX 是一个 32 位寄存器。你不能复制一个 32 位寄存器的内容到 16 位寄存器中。

在这里，暂时不必记住那些指令操作细节；当我们专门考虑那些单独的指令本身时，将深入地接触这些规则。现在只需理解一点，一些在你看来合理的事情(特别是作为一个初学者)，也许是由于技术原因违反了一些规则，可能被认为是错误的。

这些都是很简单的错误。还有一些难得多的错误，它涉及两个不同的合法源代码行之间的不一致。这里我不再举例，但是我想指出的是，错误是真正丑陋的、隐藏的东西，它需要进行很多研究，花费很多精力才能找到。总之，我们肯定不能再像在 Basic 中那么轻松了……

在不同的汇编编译器中，错误信息也有所不同，它们可能不会总是像你希望的那样为你提供帮助。比如，当遇到上面发生的“EVX” 错字时，NASM 将显示如下错误信息：

```
testerr.asm:20: symbol 'evx' undefined
```

如果你知道“符号”是什么的话，这是非常简单的。它告诉你查看哪里：20 是行号，它发现的错误地方。当你尝试加载一个 32 位寄存器的内容到 16 位寄存器中时，NASM 将向你提供没有什么帮助意义的出错信息：

```
testerr.asm:22: invalid combination of opcode and operands
```

它只是让你知道犯了对操作码及其操作数执行非法行为的错误，但仅此而已。你必须知道什么是合法的，什么是非法的，才能够真正理解做错了什么。当运行到停止标记时，对“法律”的无知不能称之为借口，汇编编译器可不像当地警察部门那样仁慈，它每次都将“逮住”你。

汇编错误信息并不能把你从理解 CPU 或汇编编译器规则的责任中免除。

从第一次坐下来编写自己的汇编代码开始，这些会变得越来越清晰。我先提醒一下你，对于更加深奥的错误，那些错误提示信息可能几乎毫无帮助价值，希望你没有被吓坏。

在编写源代码文件时，你可能制造(或将制造，让我们面对现实吧)很多错误。在这种情况下，汇编编译器将显示很多出错信息，但是，它不必将源代码文件中出现的每一个错误都一一显示。在某种程度上，多个错误交汇在一起会彻底让汇编编译器感到混乱，以至于它不一定能够再明辨是非。然而，汇编编译器一行一行地读取和翻译源代码文件，并得到了汇编语言程序在整个汇编过程中的最终“执行效果图”，这是事实。如果这幅图被拍到里面错误百出，那么最终程序将会全局崩溃。

如果已经显示了大量的出错信息，汇编程序将终止。从第一个错误开始，一定要了解它(做笔记！)，然后继续下去。如果第一个错误之后的错误不符合逻辑，那么修复第一个或前两个错误，然后再重新编译。

5.3.5 回到编辑器

修复错误的方法是把有问题的源代码文件加载回文本编辑器，然后开始仔细寻找错误。这个回路显示在图 5-9 中。它可能是你在这幅特殊地图上看到最多的公路。

汇编错误信息几乎总是包含一个行号。将光标移动到那个行，开始寻找错误和那些不切实际的幻想。如果立即发现了错误，马上修复它，然后继续寻找下一个。假设你在使用 Ubuntu 的 GNOME 图形桌面，那么保持终端窗口和编辑器窗口同时处于打开状态是有益的，这样就不必写一个行号列表到纸上或重定向编译器的输出到一个文本文件了。使用 19 寸显示器或更好的显示器，会为一次显示多个窗口提供更多的空间。

(有一种方法能让 NASM 在汇编过程中将它的出错信息写入一个文本文件中，你将在下一章中看到。)

5.3.6 汇编警告

汇编编译器可能表现得像一个沉默寡言的动物，它有时也会在汇编过程中显示警告消息。对于刚开始使用汇编语言编程的人们而言，这些警告消息几乎是个谜：它们是错误还是不是错误呢？我可以不理它们，还是应该修改源代码，直到它们不再出现为止？

唉，没有清晰的答案。对此我深感抱歉。

汇编时警告是汇编编译器作为有经验的顾问，向你暗示在源代码中有些东西有点冒险。这些东西并不严重得足以引起汇编编译器停止汇编文件，但是它的严重程度可能足以让你注意并展开调查。例如，如果你定义了一个命名标签，但是没有在它后面放置任何指令，NASM 将显示一个警告，但这可能不是错误，而是你的疏漏，你应该密切关注该行并努力回想你在编写它时究竟在想什么。(在已经距离最初编写的有问题的代码行三个星期以后，这可能并不容易。)

如果你是普通的初学者，百分之百做本书列举的每一件事情，你应该“嚼烂”汇编参考手册，弄明白为什么汇编编译器会对你发出指责的啧啧声。无视警告可能会在后来发生的程序测试中导致奇怪的错误。或者，忽略警告消息可能根本没有任何不良后果。但是我认为，知道发生了什么事情总是更好些。请遵循这条规则：

只有当确切知道汇编警告意味着什么时才可以忽略该消息。

换句话说，除非你理解了为什么会得到该条警告消息，否则，必须将它像一个错误信息一样对待。只有完全理解它的根源及其含义之后，你才可以尝试做出决定：是忽略它还是不忽略它。

总之，汇编语言开发过程的第一部分(如图 5-9 所示)是一个循环。你必须编辑源代码文件，编译它，然后返回到编辑器来修复错误，直到汇编编译器再也找不出任何错误。你不能继续进行下一步，一直到汇编编译器给你的源代码文件提供一份健康证书。我还建议研究汇编编译器提供的任何警告，一直到你已经清楚地了解了它们。修复引发警告的条件始终是一个好主意，特别是第一次开始汇编编程的时候。

如果没有进一步发现错误，汇编编译器将一个.O 文件写到磁盘中，你可以准备继续进行下一个步骤了。

5.3.7　连接目标代码文件

正如我在本章前面部分中讲到的那样，连接步骤是不明显的，而且对于新手而言，有点神秘，尤其是当只有一个目标代码模块在起作用时。但它却是至关重要的，在以前，还有可能在不需连接步骤的情况下将一个简单的 DOS 汇编语言程序直接编译成一个可执行文件，但是现代操作系统(例如 Linux 和 Windows 系统)的性质使得这样做根本没有可能。

连接步骤显示在图 5-9 的右半边。右上角是一排.O 文件。这些.O 文件是以前编译的正确的.ASM 文件，它们产生包含机器指令和数据对象的目标模块文件。当连接器(linker)连接由正在处理中的.ASM 文件所产生的.O 文件时，它将这些以前编译的.O 文件加入其中。连接器写入磁盘的单个可执行文件包含来自所有当它被调用时传递给它的.O 文件的机器指令和数据项。

一旦正在进展中的.ASM 文件完成并正确生成，它的.O 文件就能够与其他文件一起，挂到“架子”上，并添加到下一个将要处于“正在进展中”的.ASM 源代码文件中了。一点一点地，你逐渐从每一次生成和测试的模块构建出自己的应用程序模块。

一个非常重要的好处是，一些.O 模块中的过程可以被用于将来的、甚至还没有开始的汇编语言程序中。创建这样的“工具”程序库是通过重用代码节省时间的一个非常有效的方法，甚至不需要再把它传给汇编编译器！

目前，世界上有许多种汇编编译器(虽然只有少数是非常好的)，也有不少的

连接器。Linux 拥有自己的连接器，叫 ld。(这个名字其实是“load”的简写，回溯到 20 世纪 70 年代，即第一个 Unix 时代，连接器原本被叫作“loader”。)本书中比较简单的程序将使用 ld，但是到了第 12 章，我们将接纳 Linux 的特点，使用一个 C 编译器作为连接器……可以这么说。

正如我所说的，我们现在已经不再在 Basic 中了。

如同汇编编译器一样，调用连接器是从 Linux 的终端命令行完成的。连接多个文件需要在命令行中与待输出的可执行文件的名字一起，指出每一个命令文件的名字。可能还需要输入一个或多个命令行开关，以给予连接器附加的指导和说明。当你还是一个初学者的时候，这些很少能够引起你的兴趣，我将会根据需要适时对那些知识进行讲解。

5.3.8 连接错误

如同汇编编译器一样，当连接器将诸多.O 文件汇集在一起，编织成一个可执行程序文件时，可能会发现一些问题。连接错误比汇编错误要微妙一些，它们通常很难找到。幸运的是，它们比较少见，而且不容易产生。

正如汇编错误一样，连接错误是致命的，也就是说，它们使可执行文件无法生成；当连接器遇到连接错误时，它将立即终止。当你遇到连接错误时，必须返回到编辑器，找出问题所在。一旦识别出问题(或者认为已经识别出)，对源代码文件进行一些修改以修复它，你必须重新编译该程序，然后重新连接以查看连接错误是否已经消失。直到连接错误不复存在，否则你必须重返编辑器，再次尝试，然后再编译/连接一次。

如果可能，尽量通过试验和错误来避免这种情况的发生。阅读汇编编译器和连接器文档。了解你正在做什么。对在汇编编译器和连接器中发生的事情越了解，就越容易确定哪些东西适合连接器。

5.3.9 测试.EXE 文件

如果没有收到连接错误，连接器将创建一个单独的可执行文件，该文件包含出现在连接器命令行中的所有文件中的机器指令和数据项。这个可执行文件是就是你的程序。你可以在终端命令行中敲入它的名字，然后按 Enter 键来运行它，看看它能做些什么。

这里，Linux 路径开始发挥作用了，不过，如果对 Linux 稍有经验的话，你

可能已经知道这一点了。终端窗口以纯粹的文本方式来查看工作目录，并且所有这些熟悉的命令行实用程序都能够操作工作目录下的任何内容。但是，请记住，工作目录并不在路径中，除非明确地指出它，尽管人们对此有些争议，而且始终有这种争议，但是不把工作目录放入路径这一做法具有充足的理由。

从终端窗口命令行执行一个程序时，必须告诉 Linux 该程序位于何处，这可通过在该程序的名字前面的./说明符前缀来实现，它的意思是"在工作目录中"。这一点与 DOS 不同，因为在 DOS 下，任何当前目录也是可执行程序的搜索路径。在 Linux 下，一个调用程序的命令行可能是这样的：

```
./myprogram
```

这是快乐真正开始的时候。

5.3.10 错误 vs.漏洞

当你以这种方式启动程序时，会发生以下两件事情之一：

该程序将按照你的意图去工作，或者你将面对一个或更多程序漏洞(bug)的影响。漏洞是指任何不按照你预想的方式工作的事情。这使得漏洞比错误(error)更加主观。一个人可能会认为红色字符显示在蓝色背景下是一个漏洞，而另一个人则可能会认为这是一个聪明的新时代特性且让人感到非常愉悦。解决像这样的“漏洞对决特性”冲突应由你来负责。你对程序应该做什么以及怎么做应该有一个相当清楚的构想，并且应该将其备份为一套书面规范或者其他某种类型的文档，那就是判断 bug 的标准。

这种可以从视觉上即可感知的(字符的颜色奇怪)bug 很少出现。在汇编语言下工作时，更常见的情况是，bug 对于发生了什么显示很少的线索甚至没有任何提示就退出(中止)一个程序的执行。如果幸运的话，操作系统可能会打可执行文件“一巴掌”，并显示出一个错误消息。以下是一个最常见的提示：

```
Segmentation Fault
```

为了区别于汇编错误和连接错误，这样的错误被称为运行时错误(runtime error)。通常情况下，你的程序不会激怒操作系统。它只是不会做你期望它做的事情，而且很可能在失败的过程中没有说出太多的内容。

非常幸运的是，Linux 是一个坚固的操作系统，它考虑到了充满漏洞的程序，“程序打败机器”这种情况极不可能像在 DOS 时代那样常见。

话虽如此，为了阻止这种 Babel 效应，我认为这里最重要的事情是要认真地划清错误和 bug 之间的区别。错误是源代码文件中的某些错误的东西，被汇编编译器或者连接器认为不可接受而踢出来。错误阻止编译过程或者连接过程继续进行，进而阻止生成最后的.EXE 文件。

相比之下，bug 则是在程序执行中发现的问题。它既没有被汇编编译器察觉，也没有被连接器发现。bug 可以是良性的，例如在屏幕信息中拼错的单词，或在显示器上定错位的行；bug 也可以迫使程序提早中止。如果程序试图执行某些禁止的事情，Linux 将终止它，并显示出错消息。这些错误称为运行时错误，但它们实际上是由漏洞造成的。

错误和 bug 都要求你返回文本编辑器并修改源代码文件中的一些内容。所不同的是，大多数错误被报告出来，用一个行号指出你应该查看源代码的什么地方来修复该问题。相反地，bug 则被遗留下来供新手们练习。你必须寻找它们，无论是汇编编译器，还是连接器，都不会为你提供多少线索。

5.3.11 我们还在那里吗

图 5-9 表明，如果程序工作完美的话，汇编语言开发过程就可以宣告结束了。一个非常严肃的问题是：你如何知道它何时工作完美？在学习语言过程中编译的一些简单程序可能很容易在一两分钟之内测试完毕；但是任何一个能够完成某些有用事情的程序都将花费至少数小时的测试时间。一个重要的大型应用可能花费数周或者数月的时间才能进行彻底的测试。对于采用各种各样的输入值且产生各种各样的输出值的程序，要使用输入值的尽可能多的不同组合来进行测试，每次应该检查每一个可能的输出。

即使如此，在较大的程序中找到最后一个漏洞仍然被有些人认为是遥不可及的理想。也许——但是你应该以一种你可以做到的尽可能高效的方式，争取尽可能地接近这一理想。我将在本书的剩余部分对漏洞及其调试进行更多的讲解。

5.3.12 调试器和调试

汇编语言开发过程中的最后一个，也绝对是最痛苦的组成部分是调试。调试(debugging)是简单的系统过程，我们通过这一过程来定位和纠正 bug。调试器(debugger)是一种实用工具，专门设计用于帮助你定位和识别 bug。

在所有种类的软件中，调试器是最神秘、最难以理解的。调试器兼任 X 光机

和放大镜。调试器与程序一起，加载到内存中，并与程序一起，一直在内存中。然后调试器转到程序中，完成一些真正独特的事情。

调试计算机程序面临的一个问题是它们的操作速度如此之快，以至于数以万计或数十万计的机器指令可以在 1 秒钟内得到执行，如果这些指令之一不太正确，在你能够凝视屏幕识别出它是哪一个之前，它早就走远了。调试器能够让你一次执行程序中的一条机器指令，这样你就可以在每两条指令之间无限期暂停，以检查上一条被执行的指令的效果。调试器还允许你在两条指令之间的暂停期间，查看程序允许访问的内存块中的任何位置的内容以及存放在任何 CPU 寄存器中的值。

调试器能够做所有这些神秘的事情，因为它们是必不可少的，CPU 拥有一些内置到其硅芯片中的特殊属性，这些属性使调试器成为可能。它们的内部工作机制已经超出了本书的范围，但是它非常令人着迷，我建议你在熟悉了 x86 CPU 的内部结构之后，进行进一步的研究。

一些调试器能够同时显示机器指令和源代码，这样就可以看到哪些源代码文本行对应哪些二进制操作码。还有一些其他的调试器能够让你根据名字而不是简单地通过内存地址来定位一个变量。

许多操作系统都附带调试器。DOS 和早期版本的 Windows 所配备的调试器是 DEBUG，在本书的较早版本中，我详细地讲解了 DEBUG。Linux 有一个非常强大的调试器，叫 gdb，我将在下一章中连同一个单独的、用于管理它的图形化工具一起进行介绍。也有许多其他可用的调试器，我鼓励你随着技能的提高，去尝试使用它们。

5.4 沿着汇编小路旅行

你可以停止询问，“我们还在那里吗？”，此处的“那里”意思是“为创建一个实际的工作项目做好准备”。我们确实还在那里，在本章的其余部分里，我们将采用一个简单的程序，从头至尾走一遍如图 5-9 所示的过程。

你不必自己编写程序。我已经讲解了汇编语言的开发过程，但是我还没有详细讲解任何机器指令或者 CPU 寄存器。我将提供一个非常简单的程序，对于它的工作原理，我将给出足够的解释，以便使它不再是个谜。随后的章节我们将十分详细地研究机器指令及其操作。同时，你必须理解汇编语言的开发过程，否则，即便知道指令如何工作也不会对你有丝毫的帮助。

5.4.1 安装软件

关于 Linux 的诱人之处之一就是：它的可用软件多得令人目不暇接，而且几乎所有这些软件都完全免费。如果已经使用 Linux 一段时间，你就可能遇到诸如 OpenOffice、Kompozer、Gnumeric 和 Evolution 等软件。其中有些是在安装操作系统时预安装的。其余的可以通过使用软件包管理器(package manager)得到。软件包管理器是个人电脑上的一个目录程序，用于维护 Linux 下所有可用的免费软件包列表。你可以选择你想要的一款软件，软件包管理器会上网，从它们的在线主页(名为知识库)中下载它们，然后再为你安装它们。

其实，Ubuntu Linux 操作系统下安装了两款软件包管理器。一个是可以在 Application 菜单中看到的 Gnome Application Installer，它作为添加/删除菜单项而出现。这款软件包管理器由于其简单性而出现在那里，但是，它没有列出每一个你可能想要的免费软件包。藏身于 System→Administraton 菜单的是 Synaptic Package Manager，它可以(至少在理论上)访问任何一个已经被提交已知公共知识库的免费软件产品。我们将使用 Synaptic Package Manager 来获得我们需要的用作本书示例的其余软件。

不用说了，要想使用 Ubuntu 的软件包管理器，需要一个有效的互联网连接。宽带是比较好的，但是我们需要下载的两个包不是很大，所以如果相当有耐心的话，你也可以通过拨号上网连接来进行传输。

如果你以前用的是 Windows，那就更容易理解在 Linux 下，不必担心软件被安装到何处了。几乎所有软件都安装在/usr 目录层次结构下，在文件搜索路径的某个地方。你可以打开一个终端窗口，导航到任何目录，并将其作为你的工作目录，然后通过在命令行中输入一个已安装的应用程序的名字来启动该程序。

在这一章中，为了快速“游历”汇编语言的开发过程，我们需要很多开发工具：一个编辑器、一个汇编编译器、一个连接器和一个调试器，其中：

- gedit 编辑器随 Ubuntu Linux 而预安装。
- NASM 汇编编译器必须手动安装。
- Linux 连接器 ld 已经预安装。
- 调试器的情况稍微复杂一些。规范的 Linux 调试器(Gdb)是预安装的。不过，与其说 Gdb 是一个十足的调试器，不如说它是一个调试器“引擎”。为了让它真正有用(特别是对于初学者)，必须下载某种程序，从而使其更容易控制，其输出也更容易理解。这个程序叫 *KDbg*，它是 Gdb

的一个“前端”。我将在下一章解释它如何工作。现在，只需无条件地相信它即可。

Synaptic Package Manager 能够让你在一次操作中选择安装多个软件包。打开 Synaptic。它将刷新其索引数秒钟，然后显示一个如图 5-10 所示的窗口。

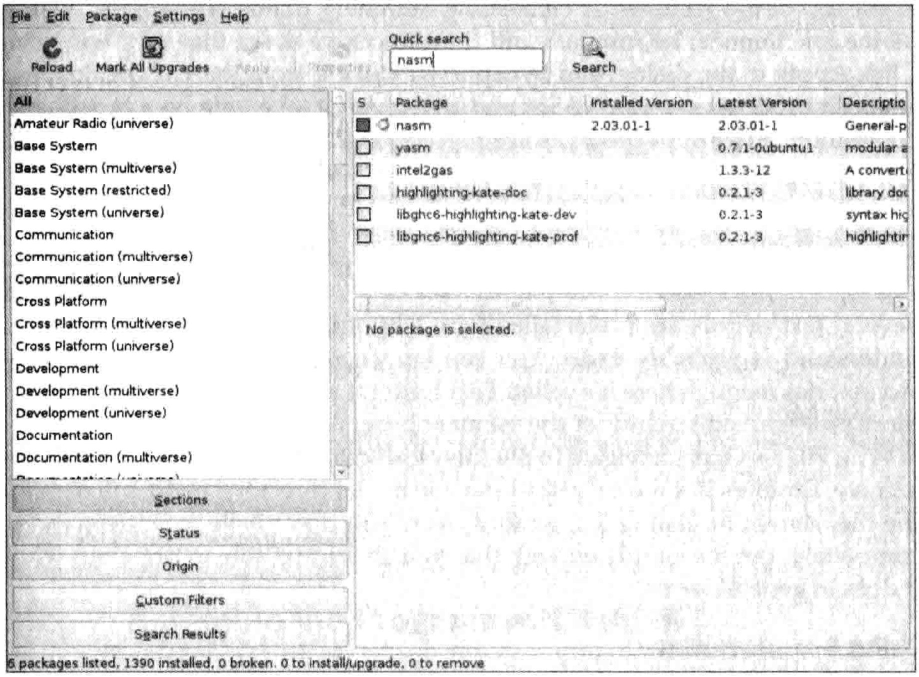


图 5-10　Synaptic Package Manager

单击 Quick Search 域，使其获取焦点，然后输入 NASM。不必单击“Search”按钮；因为在你敲入关键字时，Synaptic 会进行渐进式搜索，并且显示它认为符合你输入的搜索文字的任何软件包。NASM 应该是显示结果中的第一项。在结果窗格中的左侧边缘是一列复选框。选中 NASM 对应的复选框。将会弹出一个确认对话框，从中选择用于安装的标记。现在，根据 NASM 的版本的不同，可能会出现下一个对话框，询问你：“标记额外需要的更改吗？”这些是指安装该产品所需的额外内容(通常是指代码库)，因此单击“标记”按钮。一旦确认额外的更改，NASM 行将变成绿色，表明 NASM 已被放入安装队列。NASM 需要的所有文件也将变成绿色。

为 KDbg 做同样的事情。在快速搜索中敲入 KDbg，并且当复选框出现时，

启用它。它也需要安装额外的库文件，所以当相应对话框出现时，单击 Mark 按钮。

所有需安装项的所在行都变为绿色之后，单击窗口顶部的 Apply 按钮。一个确认概要对话框将会出现，列出所有即将安装的单项任务及安装它们所需的硬盘空间。单击对话框中的 Apply 按钮，Synaptic 将从那里拿出它。下载如 NASM 和 KDbg 一样小巧的产品不会花费太多的宽带连接时间。当这两种产品都出现在电脑中时，Synaptic 将安装一切，当出现 Changes Applied 对话框时，大功告成！

5.4.2　第 1 步：在编辑器中编辑程序

Ubuntu 里预安装了几个文本编辑器，它们之中最容易理解的可能就是 gedit 了。可以通过 Applications→Accessories 菜单来启动它，在那里它被叫作文本编辑器。后面，我将介绍 Kate，它是一款更加强大的编辑器，但暂时我们先使用 gedit。

通过 File→Open 菜单项，从本书的存档列表中浏览到 eatsyscall 目录。双击该目录下的 eatsyscall.asm 文件。gedit 将显示该文件，如清单 5-1 所示。将它通读一遍。你不必完全理解它，但是它相当简单，你应该完全能够大致预测出它在整个过程中的作用。

清单 5-1：eatsyscall.asm

```
; 可执行程序名 : EATSYSCALL
; 版本  : 1.0
; 创建日期  : 1/7/2009
; 最后修改日期 : 1/7/2009
; 作者  : Jeff Duntemann
; 描述  : 一个简单的 Linux 下的汇编应用程序，使用 NASM 2.05，
;    演示了 Linux INT 80H 系统调用显示文本的用法。
;
; 使用这些命令生成：
; nasm -f elf -g -F stabs eatsyscall.asm
; ld -o eatsyscall eatsyscall.o
;

SECTION .data                  ; 包含已初始化的数据的段

EatMsg: db "Eat at Joe's!",10
EatLen: equ $-EatMsg
```

```
SECTION .bss                    ; 包含未初始化的数据的段
SECTION .text                   ; 包含代码的段

global _start                   ; 连接器需要据此找到入口点!

_start:
        nop                     ; 这个无操作指令让 gdb 非常高兴 (见正文)
        mov eax,4               ; 指定 sys_write 系统调用
        mov ebx,1               ; 指定文件描述符 1: 标准输出
        mov ecx,EatMsg          ; 传递显示信息的偏移地址
        mov edx,EatLen          ; 传递显示信息的长度
        int 80H                 ; 进行系统调用来输出文本到标准输出
        mov eax,1               ; 指定 Exit 系统调用
        mov ebx,0               ; 返回一个零代码
        int 80H                 ; 进行系统调用来终止程序
```

此时，你可以修改源代码文件(如果想这样做的话)，但现在只是简单地一下读它。它属于一种名为“Hello world!”的演示程序，只是简单地在一个终端窗口显示单行文本。(你必须从某个地方开始！)

5.4.3 第 2 步：使用 NASM 编译程序

NASM 汇编编译器并不具有非技术人员理解中的、当今意义上的图形用户界面。它没有窗口，根本没有地方可以输入文件名或者选择复选框选项。NASM 只通过文本进行工作，你得通过终端和 Linux 控制台和它进行交流。这就像是在那些古老的 DOS 时光，一切都要通过命令行输入。(我们忘得多么快呀！)

打开一个终端窗口。在 Ubuntu Linux 操作系统下可以使用许多种不同的终端实用工具。我用得最多的一款终端实用工具叫 Konsole，但它们都可用于这里。终端窗口打开时，一般会以 home 目录作为你的工作目录。一旦得到了命令提示符，使用 cd 命令导航到“eatsyscall”项目目录：

```
myname@mymachine:~$ cd asmwork/eatsyscall
```

如果你是一个 Linux 新手，一定要通过使用 ls 命令检查目录的内容，以确保正处于正确的目录下。至少文件 eatsyscall.asm 应该在那里，或者是从本书的存档列表中导出的，或者由你输入到一个文本编辑器中的。

假设 eatsyscall.asm 文件存在，(小心地)敲入如下命令并按 Enter 键来编译它：

```
nasm -f elf -g -F stabs eatsyscall.asm
```

如果 NASM 没有发现问题，它将什么也不说，你会回到命令提示符。这意味着，编译成功完成！如果你是自己输入的 eatsyscall.asm 文件，并输入了某些不正确的东西，你可能会收到一个错误。请确保该文件与清单 5-1 一致。

那么，所有输入到终端的内容含义是什么？我已经在图 5-11 中剖析了刚才输入的命令行。一个 NASM 调用从该程序本身的名字开始。之后的一切都是控制编译过程的参数。这里显示的几乎是所有初次学习汇编语言开发过程时可能需要的命令行。此外，还有一些具有更多神秘目的的命令行，NASM 文档中对所有它们进行了总结。下面依次介绍这里使用的命令：

-f elf: 有许多种有用的目标文件格式，而且每一种都以不同的方式生成。NASM 汇编编译器能够产生其中的大多数，包括其他格式，如 bin、aout、coff 和 ELF64，你可能不需要它们，至少暂时不需要。-f 命令告诉 NASM，它将对目标代码文件采用哪种格式。在 32 位 IA – 32 Linux 工作环境下，格式为 ELF32，它可以在命令行中通过简单的 elf 来指定。

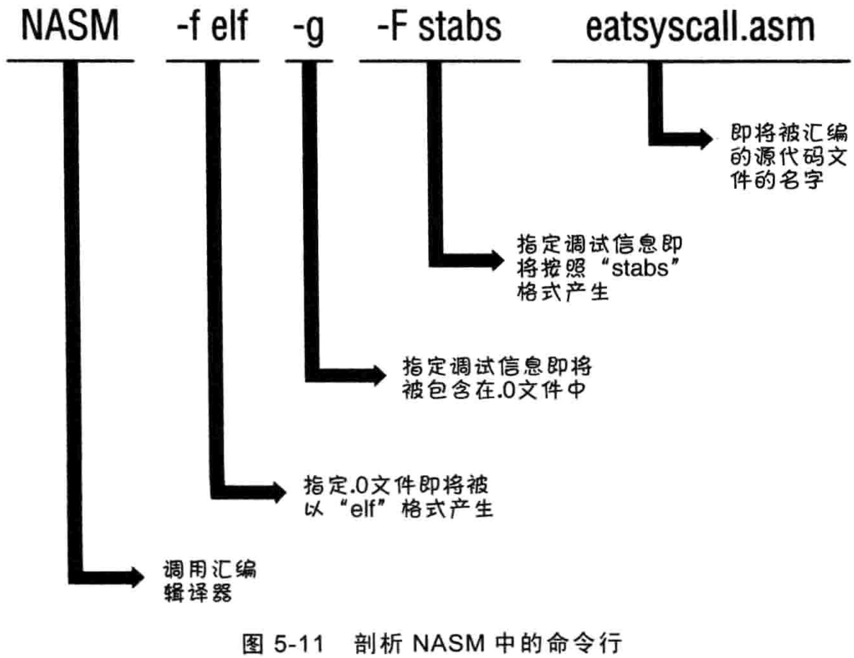


图 5-11　剖析 NASM 中的命令行

-g: 当你仍然忙于一个程序时，你想让调试信息嵌入目标代码文件，以便可

以使用调试器来定位出现问题的地方。(很快会有更多关于如何做到这一点的讲述。) - g 命令告诉 NASM 将调试信息包含到输出文件中。

-F stabs: 与输出文件一样，NASM 也可以采用不同的格式生成调试信息。再次地，与输出文件格式一样，如果你工作于 IA-32 Linux 下，很可能为调试信息使用 stabs 格式，至少在你刚刚开始学习汇编时。还有一种更为强大的调试格式，叫 DWARF，也可与 elf 一起使用(弄明白了吗？)，如果在该条命令中将“stabs”替换为“dwarf”，NASM 将生成 DWARF 数据，而不是 STABS 数据。还请记住，Linux 命令是区分大小的。-f 命令和-F 命令是不同的，所以一定要监视 Shift 键！

eatsyscall.asm: 在 NASM 命令行的最后一项，始终是即将汇编的文件的名字。同样，与 Linux 中的所有事情一样，文件名都是区分大小的。EATSYSCALL.ASM 和 EatSysCall.asm(以及所有其他大小写变化形式)被认为是完全不同的文件。

除非给了 NASM 其他命令，否则它会生成一个目标代码文件并使用源代码文件的名字为其命名，文件的扩展名为.O。这些“其他命令”可通过-o 选项给出。如果在 NASM 命令行中加入-o 命令，后面必须跟一个文件名，该文件名就是你希望 NASM 为生成的目标代码文件赋予的名字。例如：

```
nasm -f elf -g -F stabs eatsyscall.asm -o eatdemo.o
```

在这里，NASM 将把源文件 eatsyscall.asm 编译为目标代码文件 eatdemo.o。

现在，在进入连接步骤之前，请通过使用 ls 命令列出工作目录的内容，验证一下目标代码文件是否已经被创建。文件 eatsyscall.o 应该已经在那里了。

5.4.4 第 3 步：使用 LD 连接器

到目前为止，一切顺利。现在，你必须使用 Linux 连接器实用工具(ld)创建一个可执行程序。确保目标代码文件 eatsyscall.o 出现在工作目录之后，在终端中键入下面的连接器命令：

```
ld -o eatsyscall eatsyscall.o
```

如果最初的程序通过编译过程且没有任何错误或警告，该目标文件应该也能不带任何错误地通过连接过程。如同在 NASM 中一样，当 ld 没有遇到值得一提的内容时，它就什么也不说。在汇编语言世界中，没有消息就是最好的消息。

用于连接的命令行比用于编译的命令行简单，如图 5-12 所示。“ld”运行连接器本身。-o 命令指定一个输出文件名，这里为 eatsyscall。在 DOS 和 Windows 世界里，可执行文件几乎总是使用.exe 作为文件扩展名。在 Linux 世界里，可执行文件一般情况下根本没有扩展名。

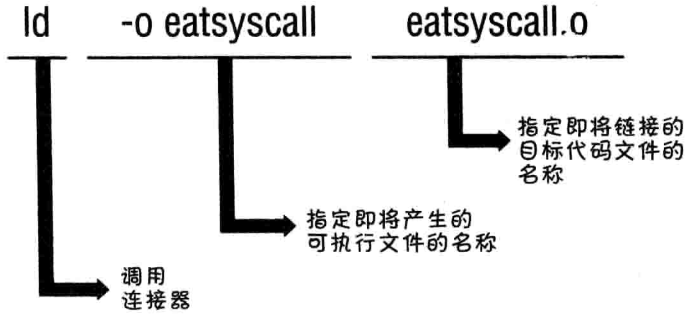


图 5-12　剖析一条 ld 命令行

请注意，如果不使用-o 命令指定可执行文件的名字，ld 会创建一个默认名为 a.out 的文件。如果你曾经在项目目录中看到过一个神秘的名为 a.out 的文件，它可能意味着你没有使用-o 命令运行连接器。

在 ld 命令行中最后输入的内容是即将被连接的目标文件的名称。在本例中只有一个，但是一旦开始使用代码库(不论是自己编写的还是别人编写的)，必须在命令行中输入正在使用的每一个库的名字。输入顺序并不重要，只要确保它们都在那里即可。

5.4.5　第 4 步：测试可执行文件

一旦连接器无错通过，已完成的可执行文件将在工作目录下等着你。如果汇编编译器和连接器消化了它，且没有显示任何错误消息，表明它没有错误。然而，没有错误并不意味着没有缺陷。为了确保其工作，只需在终端上命令行上指出其文件名：

```
./eatsyscall
```

Linux 新手需要记住，你的工作目录并不会自动出现在搜索路径上，如果只是在命令行中键入可执行文件的名称(没有代表“工作目录”的前缀“./”)，Linux 将找不到它。但是，如果名字前面有那个前缀，可执行文件将加载和运

行，并打印出它的由 13 个字符组成的广告：

```
Eat at Joe's!
```

胜利完成任务！但是，撤离终端窗口还不是时候……

5.4.6 第 5 步：在调试器中观察程序运行

假设清单 5-1 输入正确(或者它是从存档清单中解压的)，在 eatsyscall.asm 中就不会有错误。对于程序员而言，这是一种极为罕见的情况，特别是那些刚刚起步的程序员。大多数时候，几乎需要立即开始查找漏洞。为此，最简单的方法是将可执行文件加载到调试器中，以便进行单步处理，在每条机器指令执行完毕之后，暂停一下，看看每条指令对寄存器和内存中定义的每一个变量有什么影响。

gdb 和 KDbg 两个程序一起，向你提供一个愉快的(嗯，可容忍的)调试经历。gdb 实用程序完成调试器所做的纵深层次的 CPU 魔术，而 KDbg 则在显示器上将其排放整齐，并允许你控制它。要想启动一个调试会话，从终端窗口命令行中调用 KDbg，后面跟可执行文件的名称：

```
kdbg eatsyscall
```

KDbg 并不像 NASM 和 ld 那样几乎不出声。它是一个 KDE 应用程序，搭建了一个很好的图形窗口，看起来非常像图 5-13 中所示的界面。eatsyscall 程序的源代码应该在中间的窗格显示。如果顶部窗格的左上角没有显示“寄存器”，请选择“查看”→“寄存器”菜单项，并确保寄存器项的旁边有一个 X。

为了腾出更多的空间来查看源代码，可以关闭 KDbg 的底部窗格。单击窗格右上角的小 X。KDbg 能够在不同的窗口中显示很多不同的事情，但是为了快速处理，显示源代码窗格和寄存器窗格就足够了。让其他的窗口处于打开状态只会给你添乱。

正如本章前面所述，在要求调试器开始单步调试一个程序之前，它们必须被告知最初停在哪里，否则，它们通过整个程序的执行速度太快了，你根本跟不上。所以必须设置一个初始断点。实现这一点的方法是滚动源代码的显示内容，直到在程序文本左侧边缘处可以看到标签_start：之后的代码。下移 2 行，在源代码窗格左侧边缘的空白处，在窗口的框架和加号之间的位置上，按一下鼠标左键。当你点击时，应该会出现一个红点。这个红点表明你已经在那行代码设置了断点，在这本例中，该行代码是指令：MOV EAX，4。(请确保在这条指令而不

是程序中刚好位于它上面的那条 NOP 指令插入一个断点！)

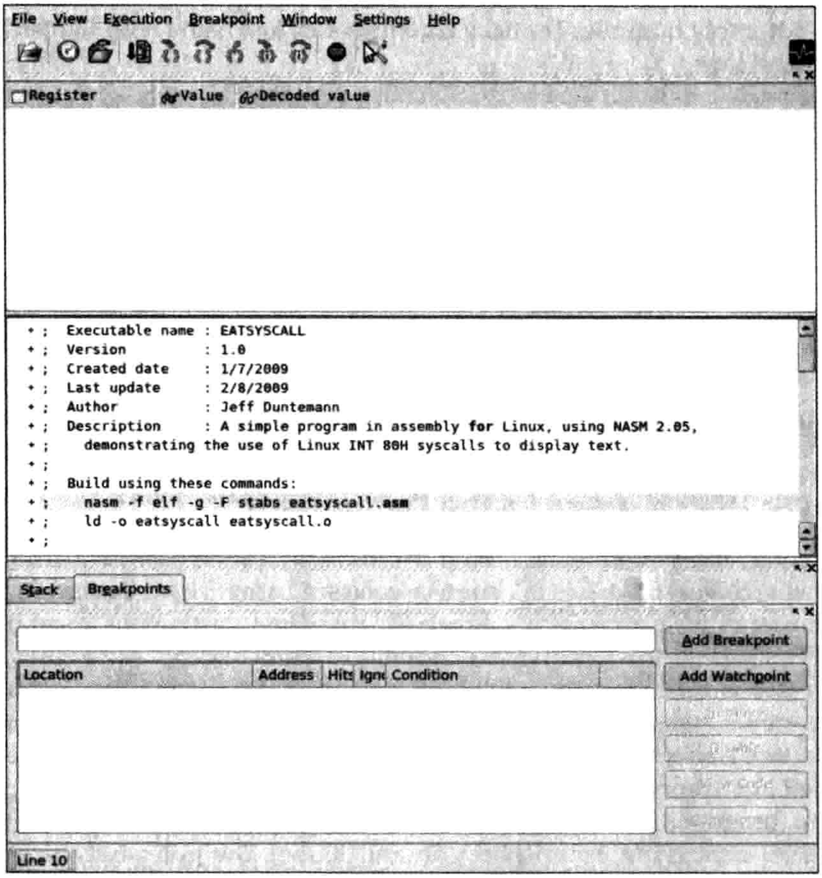


图 5-13　KDbg 的启动窗口

一旦已经设好了初始断点，单击位于顶部工具栏的 Run 按钮。该按钮看起来像一个左边有向下箭头的页面。(当鼠标指针悬停于这个按钮时，它应该显示“运行”两个字。)几乎在同一瞬间发生两件事情(参见图 5-14)。

- 红点显示的断点将被一个指向右侧的绿色三角形所覆盖。三角形表明程序中已暂停执行的地方，同时它还指向下一条即将被执行的指令。注意，由于断点而暂停执行的那个位置上的指令并未得到执行。)
- 之前是一片空白的顶部窗格，现在充满了一系列 CPU 寄存器。它是一个长长的列表，因为它包括了所有的 CPU 标志寄存器和浮点处理器寄存

器，但是你只需要观察这个演示示例的顶端那组寄存器。



图 5-14 KDbg，单步调试准备就绪

这时候，通用寄存器都显示为包含零。程序已经开始运行了，但是唯一一条已经运行的指令是 NOP，它什么都不做。(这是一个占位符，为什么它会出现在程序的这个地方，要等到下一章进行讲解。)

这种情况将很快会发生改变。为了做第一个单步调试，单击 Step Into By Instruction 按钮。依次将鼠标指针悬停在那些工具栏的按钮上，直到你找到它。(该按钮在图 5-14 中已经获得了焦点。正如按钮名称所建议的顾名思义，单击该按钮会触发一条机器指令的执行。那个绿色三角形将向下移动一行。)

在屏幕上方的寄存器窗口中，情况也与以前不尽相同。有两行已变成红色。红颜色表明，在我们刚才看到的单步执行过程中，它们的显示值已经发生了变化。EIP 寄存器是指令指针寄存器，它跟踪下一条将要被执行的指令。现在，更有趣的是 EAX 寄存器的状态。前面看到的 0x0 现在变成了 0x4。如果查看源代码，我们会发现，刚才执行的是这条指令：

```
mov eax,4
```

“MOV”助记符告诉我们，数据正在被移动。左边的操作数是目标操作数(数据移到何处)，右边的操作数是源操作数(数据来自何处)。发生的事情就是：该条指令把数值 4 放到了寄存器 EAX 中。

再次单击 Step Into By Instruction 按钮。该指针将再次向下移动一行。再次地，寄存器窗口中的红色的行表明正在执行的指令所改变的内容。指令指针寄存器再次改变，这应该是意料之中的事情。我们每次执行一条指令时，EIP 都是红色。这一次中，EAX 已经再次变成黑色，而 EBX 已经变成红色。EBX 的值已经从 0 变成 1。(符号“0x1”只是另外一种告诉我们该值是以十六进制表示的方式。)显然，我们已经把数值 1 放入寄存器 EBX，这就是我们刚才执行的指令：

```
mov ebx,1
```

再次单击该按钮。这一次，寄存器 ECX 的值完全改变(见图 5-15)。每个人在自己的个人电脑上看到的 ECX 的确切值跟我截屏时看到的可能会有所不同。这个值取决于个人的 Linux 系统，拥有多少内存以及 Linux 操作系统在内存中其他地方正在做什么。重要的是一个 32 位的十六进制值已被移到 ECX 中。下面这条指令完成该项任务：

```
mov ecx,EatMsg
```

那么，我们实际上移动的是什么内容呢？如果暂时向上滚动到较前部分的源代码，你会发现 EatMsg 是一个带引号的字符串，里面是普通的字符，其内容为“Eat at Joe's! ”，而不是一个 32 位数字，但注意指令右侧的注释：“传递消息的偏移地址。”我们其实加载到 ECX 的不是消息本身，而是该消息在内存中的地址。从技术上讲，在基于 IA – 32 的保护模式下，像 EatMsg 这样的数据项既有段地址，也有偏移地址。然而，段地址是操作系统的财产，当作这种简单的用户空间编程时，我们可以放心地忽略它。回到 DOS 时代，我们则必须使用实模式分段内存模型，而且必须跟踪段寄存器；采用保护模式的方式来完成它意味着少

了一个令人头痛的问题。(别担心，还有很多方法！)



图 5-15　移动一个内存地址到寄存器中

再次单击 Step Into By Instruction 按钮，寄存器 EDX 将赋值为 0xe，或(十进制)14。它是 EatMsg 字符串的长度。

此时，所有关于将不同的值移动到其目标位置的设置工作已经完成。单击按钮，执行下一条指令：

```
int 80H
```

看起来好像什么也没有发生——寄存器窗口中什么也没变——但是，请等一等。打开 KDbg 的菜单，并选择 View→Output 子菜单。一个简单的终端窗口将会出现——而且有 EatMsg，告诉世人去哪里吃午饭(见图 5-16)。

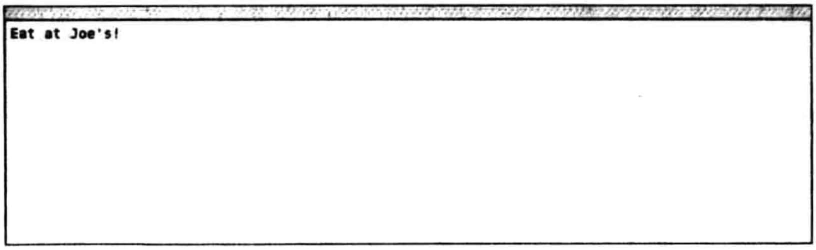


图 5-16　终端窗口中的程序输出

INT 80H 指令是一条特殊的指令。它生成一个名为 sys_write 的 Linux 系统调用(我们亲切地称之为 *syscall*)，它将数据发送到当前活动的终端窗口。

将 EatMsg 发送到输出窗口，这就是 eatsyscall 程序的设计初衷。它的任务完成后，程序中的最后三条指令主要做一些整理工作，然后结束。单击一下按钮，单步通过它们来进行观察，务必使 EAX 与 EBX 在最后的 INT 80H 前接收到新值，这些新值向 Linux 发出信号，告诉它该程序已经完成。在 KDbg 窗口底部的行中，你将得到一个对它的确认，显示“程序正常退出”，以及退出发生的源代码行的行号。

你可能遇到的一个问题是：为什么步进按钮叫“Step Into By Instruction”？我们只是往下进入下一行，并没有步入任何地方呀。必须等我们进入有关过程的讲解之后，才能有一个完整的答案，但是它的要点是：KDbg 让你选择是一步一步地跟踪一个汇编语言过程的执行，还是当执行该过程时让计算机全速运行。Step Into By Instruction 按钮指明一步一步地通过一个过程。“Step Over By Instruction”(不进入方法调试)按钮(右侧的下一个按钮)则允许过程全速执行，然后再捡起该过程调用之后的单步调试。

为什么跨过(而不进入)一个过程调用呢？主要是因为：过程通常是库过程，它可能是你或者其他人几个月或几年前编写的。如果它们已经通过了调试，并且工作正常，再次通过它们更可能是浪费时间。(但是，通过让它一次运行一条指令来观察某个过程的运行情况，可以知道很多关于该过程的工作机理。)

反过来说，如果对于现有程序而言是新程序，我们可能需要步入它们，而且像步入程序的主要部分那样仔细。KDbg 提供了这个选项。简单的程序中不包含任何过程，因此，Step Into By Instruction 按钮和 Step Over By Instruction 按钮做同样的事情：执行序列中的下一条指令。

Step Into By Instruction 按钮左侧的三个单步调试按钮用于调试高级语言代码，如 C 语言。它们允许一次步入一条高级语言语句，而不是一条机器指令。这

些按钮并不适用于汇编语言编程，所以这里不再对其作进一步的讨论。

5.4.7 准备好要来真格的了吗

我已经以极其缓慢的速度讲解了这一章；如果你已失去耐心，现在可能会痛苦不堪地呻吟了。忍耐一下。我想确保你非常清楚地了解汇编语言编程过程中的这些组成步骤。到目前为止，本书中所有内容都是概念性的基础，但是此时此刻，你已经具备了必要的基础。现在是时候拿出一些正式的工具，调查程序员对 Linux 操作系统的看法，并开始编写一些程序了。

第6章

有地儿，有工具

与工作息息相关的Linux 操作系统和工具

阿基米德，这位早期的工程师，有一句脍炙人口的名言："给我一个足够长的杠杆和一个支点，我将撬起整个地球。"老先生没有给出多少比喻，字面上说的是利用真正的长杠杆的机械优点，但是在他的话背后，却隐藏着一个关于一般工作的更大的真理：要想完成某件事情，需要有一个工作的地方，而且该地方需要有可用的工具。在我的地下室里，无线电工作台是以这种方式设置的：一个大而平的空间用来放置效果不太好的发射台，在它上面的货架上放置示波器、VTVM、频率计数器、信号发生器、信号跟踪器和栅陷测试振荡器，它们的取拿都很方便。在只有两步之遥的对面墙上，有一长排货架，我在它上面放些零件(包括我的传奇收藏——电子管)、原材料(如板材、电路板原料和塑料废料)以及那些不常使用的设备。

从某些方面来讲，在完成计算工作期间，操作系统就是你工作的"场地"。所有需要的工具都应该位于一个触手可及的地方，而且应该有一个标准的、容易理解的访问方式。数据存储应该位于"附近"，并且易于浏览和查找。Linux 操作系统满足了这一需求，在当今的桌面计算世界中，几乎没有其他操作系统能够做到这一点。

古老的操作系统，如 DOS，以一种"有限"的方式为我们提供了"立足之地"。DOS 提供对磁盘存储的访问，并且以一种标准的方式加载和运行软件，仅此而已。工具集很小，但它却是一个很好的开端，我们可以在 6 MHz 的 8088 机器上对所有的工具进行管理。

在某些方面，关于 DOS 最有趣的事情是，它被创建为更像是某个更强大的操作系统(即 Unix 系统)的"简化"版本，Unix 系统是 AT&T 研究实验室在 20 世

纪 60 年代和 70 年代开发的。当时，IBM 个人电脑已经出现，Unix 只运行在硕大而昂贵的大型机上。个人电脑没有原始的计算能力来运行 Unix，而 DOS 被创建为具有一个与 Unix 文件系统非常相似的分层文件系统，可通过它的命令行访问那些工作方式与 Unix 工具非常类似的工具子集。

经过多年的发展，x86 电脑已经不断成长，到了 1990 年左右，Intel 的 CPU 已经非常强大，足以能够运行类似于 Unix 的操作系统了。个人电脑“向上增长”，Unix“向下增长”，直到到两者在中间碰了面。1991 年，年轻的芬兰程序员 Linus Torvalds 编写了类似于 Unix 的操作系统，它能够运行在廉价的 386 个人电脑上。该系统基于一个名为 Minix 的 Unix 实现，Minix 是一个与 Unix 外形相似且能够运行在小型计算机上的操作系统，写于 20 世纪 80 年代的荷兰。Torvalds 的 Linux 操作系统最终逐渐主宰了 Unix 世界。Unix 的其他桌面变体出现了，最重要的是 BSD(伯克利软件发布)的 Unix，它产生了一些小型系统的实现，并最终成为苹果公司的 Mac 的 OS / X 操作系统的核心组成部分。

这就是我们站立的地方，而且是一个很好的地方，就工具而言，它还拥有专门为我们现在所做的工作类型而设计的某种软件工作台。NASM 汇编编译器非常强大，但是“沉默寡言”，并且不可避免地与命令行联系在一起，就像你能够在 Linux 上找到的大多数古老的 Unix 工具一样。在前面的第 5 章中，我们以一种老式的、比较困难的方式，通过在命令行键入命令，匆匆完成了一个简单的开发项目。你需要知道它的工作过程，但它绝不是我们能够做的最佳方式。

20 世纪 80 年代，DOS 下的 Turbo Pascal 取得了传奇式的成功，主要是因为它把编辑器和编译器集成在一起，并且提供了一个菜单，使我们能够方便快捷地在(用以编写代码的)编辑器、(用以将代码编译成可执行文件的)编译器以及(能够运行和测试这些文件的)DOS 之间移动。在 Turbo Pascal 下编程比较容易掌握，而且比传统方法更加快速，后者需要不断从命令行发出命令。

Turbo Pascal 是第一个真正成功的、能够为程序员提供交互式开发环境(Interactive Development Environment，IDE)的商业产品。其他的虽然很早就出现了(尤其是最初的 UCSD p-System)，但只有 Turbo Pascal 真正做到了让这个想法为大家所了解。

6.1 Kate 编辑器

有一点值得注意，即在 Linux 下的汇编语言领域中，并没有真正等同于 Turbo Pascal 的集成环境。其理由在初学者看来，似乎比较独特：经验丰富的汇编语言程序员们或者创建自己的开发环境(毕竟，他们是编程精英)，或者干脆在赤裸裸的操作系统命令提示符下工作。Turbo Pascal 类型的 IDE 对于他们的吸引力，远不如像对你那样强烈。然而，诸如 C 语言、C++和 Python 之类的高级语言编程都有一种向 IDE 靠拢的趋势。我认为汇编语言程序员最终也会赞成的。(两个值得关注的通用集成环境是 Eclipse 和 KDevelop，尽管目前二者都还没能用于汇编语言。)

与此同时，在你学习汇编语言期间，我将介绍一款简单的 IDE 供你使用。它具有朴素的美德，而且适用于编写那类入门级的小型程序，我没有发现任何一款像它一样的其他开发环境。它叫 Kate，并且自 2002 年以来，日渐流行。作为文本编辑器，Kate 因以下诸多原因而不同寻常。

- 它能够识别汇编语言源代码的格式，并以不同的颜色突出显示不同的汇编语言源代码元素。
- 它包括管理和编辑文本文件的功能，这通过一个侧面的窗格来实现。
- 它的较大的 MDI(多文档界面)显示中集成了一个终端窗口，可以在 MDI 中启动程序进行测试，也可以启动调试器进行单步调试，执行内存检查。
- 它为项目提供会话(session)支持，会话就是当你工作于某一特定项目时 Kate 编辑器所处的“状态”。
- 它可作为软件组件来使用；通过使用 KDE 的 KParts 技术，你可以将它构建到自己编写的程序中。

事实上，Kate 编辑器是“大型”IDE Kdevelop 和类似于 Access 的数据库管理器 Kexi 中使用的编辑组件。尽管 Kate 起源于 KDE 世界并依赖于 Qt 代码库，但是它完全可以轻松地在基于 GNOME 的 Ubuntu Linux 版下安装和运行。

6.1.1 安装 Kate 编辑器

如果你正在使用 Ubuntu Linux 操作系统，那么可以非常容易地从 Applications→Add/Remove 菜单项安装 Kate 编辑器。在“搜索”域中输入

“**kate**”，它就会出现了。选中安装复选框，确认安装它所需的依赖项，软件包管理器将完成剩余的工作。在选择 Kate 编辑器之后，返回并搜索 KWrite 编辑器，选中相应的安装复选框。KWrite 是一款简洁的编辑器，与 Kate 编辑器基于相同的编辑引擎；同时它本身也是有用的，为了安装 Kate 的插件，应该安装 Kwrite。奇怪的是，Kate 编辑器本身并没有安装自己的插件，更加奇怪的是，KWrite 编辑器不能使用它为 Kate 编辑器安装的插件。(没有一个人说过这事儿符合常理！)

6.1.2　启动 Kate

安装以后，可以从 Ubuntu 的 Applications 菜单的 Accessories 组中启动 Kate 编辑器。如果习惯于将图标放到桌面的顶级面板，可以把 Kate 编辑器的图标也放在那里。这样做即可实现该效果：选择 Applications→Accessories→Kate 菜单项，但在左击 Kate 打开它之前，先右击它。将会出现一个上下文菜单，其中最上面的那个菜单项就是“Add this launcher to panel”(将此启动程序添加到面板)。右击此菜单项，Kate 编辑器的图标便被放到顶部面板。参见图 6-1，该图中我已经在面板中放了它的图标，所以你可以看到它出现在哪个地方：就在“帮助”按钮的右侧。

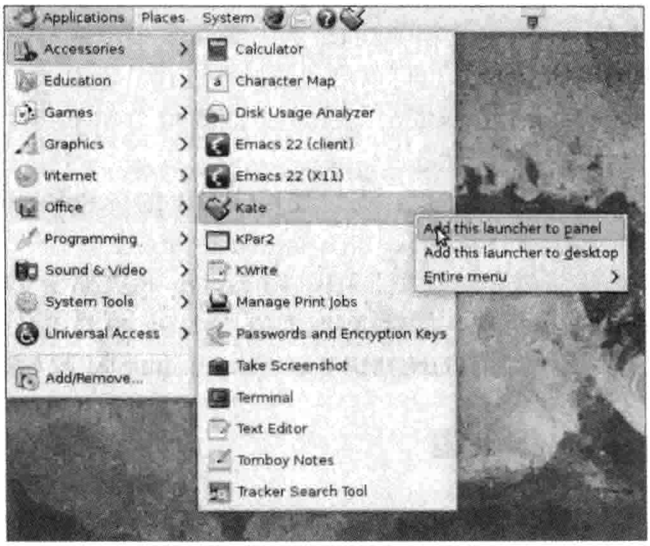


图 6-1　把 Kate 编辑器的图标放到面板上

与面板图标相比，如果你更喜欢桌面图标，同样的上下文菜单还显示了一个菜单项，让你能够把 Kate 编辑器的启动图标放到桌面上。至于具体放在桌面的什么位置，由你决定。

我不建议通过终端命令行启动 Kate 编辑器。这样做是可以奏效的，但是当 Kate 运行时，终端会显示对于 Kate 编辑器程序本身的调试信息。这是一个 bug 还是一个特性取决于你是哪种用户，但是终端窗口中的所有活动都会令你分心，而且与你正在处理的项目毫不相关。

第一次启动 Kate 编辑器的时候，在看到该编辑器本身的屏幕之前，你会看到一个空的 Session Chooser(会话选择)对话框(见图 6-2)。由于还没有创建任何会话，所以没有会话被列出。理解会话的定义及其工作机制(稍后将深入介绍)后，可以创建一个新的会话，但是目前，单击 Default Session(默认会话)，让它高亮显示，然后单击 Open Session(打开会话)按钮。Kate 编辑器的主窗口将会出现，上面覆盖着一个 Tip of the Day(日期提示)窗口。有些人觉得这样的“启动提示”对话框非常讨厌，但是在第一次学习 Kate 编辑器时，它们可能是有用的。如果不希望看到它们，取消选中 Show Tips on Startup(在启动时显示提示)复选框即可。

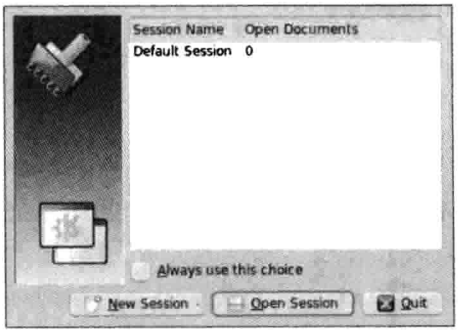


图 6-2　会话选择对话框

Kate 编辑器的默认屏幕布局(见图 6-3)很简单。编辑器窗格在右侧，管理器工具条在左侧。注意，如果在管理器工具条的左边距处没有看到 Filesystem Browser(文件系统浏览器)按钮，在底部边缘也没有看到 Terminal(终端)按钮，通常意味着你还没有安装 KWrite 编辑器，它会安装一些插件供 Kate 编辑器使用。其中包括终端插件，这对生成和运行本书提供的示例来说，是必不可少的。请确保已经安装了 KWrite 编辑器，否则你的 Kate 编辑器的安装并不完整！

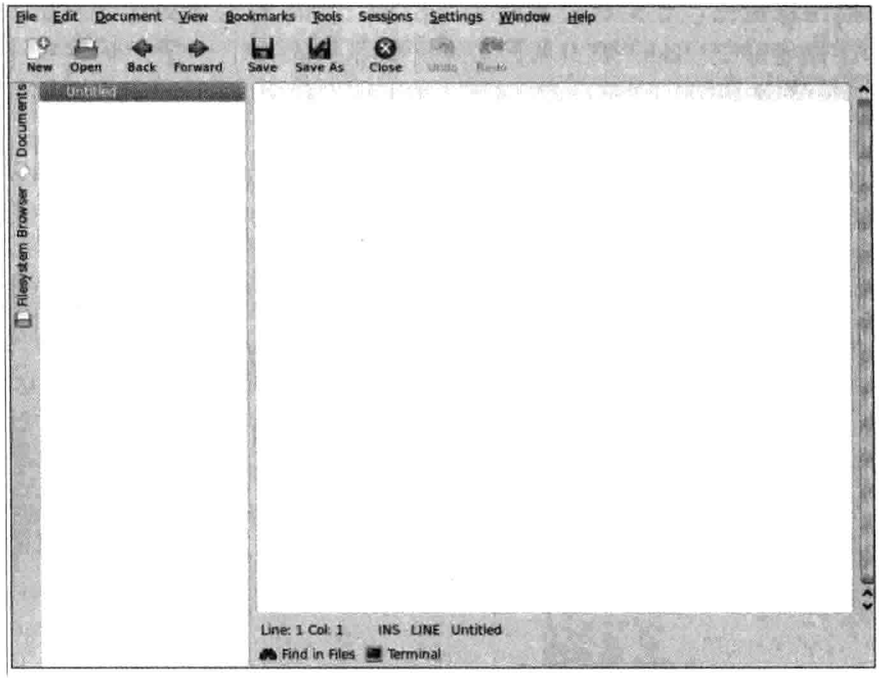


图 6-3 Kate 编辑器的默认屏幕布局

6.1.3 配置

Kate 编辑器支持大量的可配置选项。其中一些比其他的更有用，而实际上大多数并不真正适用于汇编语言编程。那些真正适用的选项会使你的编程生活变得更加容易。安装 Kate 编辑器之后，应该设置如下所示的选项。

- **编辑器模式(Editor Mode)**：选择 Tool→Mode→Assembler→Intel x86 (**NASM**)。这有助于 Kate 编辑器识别 NASM 中的语法。
- **高亮显示语法(Syntax Highlighting)**：选择 Tool→Highlighting→Assembler→Intelx86 (**NASM**)。这使得 Kate 编辑器在编辑器窗口中高亮显示源代码助记符和符号。
- **启动行号显示(Enable line number display)**：打开 View 菜单，单击 Show Line Numbers 菜单项前面的复选框。也可以通过 F11 功能键来切换显示行号和关闭显示行号。我们可以通过行号查阅程序内部的单独代码行。
- **启动图标边框显示(Enable the icon border display)**：选择 View→Show

Icon Border。这是(用星号)显示书签的地方；如果打算在文件中使用书签，图标边框必须是可见的。注意，任何时候都可以通过按 F6 键切换图标边框的显示和关闭。

- **启用外部工具(Enable external tools)**：选择 Settings→Configure Kate，单击选项树中的 plugins 插件行。这将弹出 Kate 的 Plugin Manager(插件管理器)对话框(见图 6-4)。安装 Kwrite 编辑器后，以下三个插件应该已经启用：Terminal tool view、Find in files tool view 和 File system browser。找到并选中 External Tools 复选框。

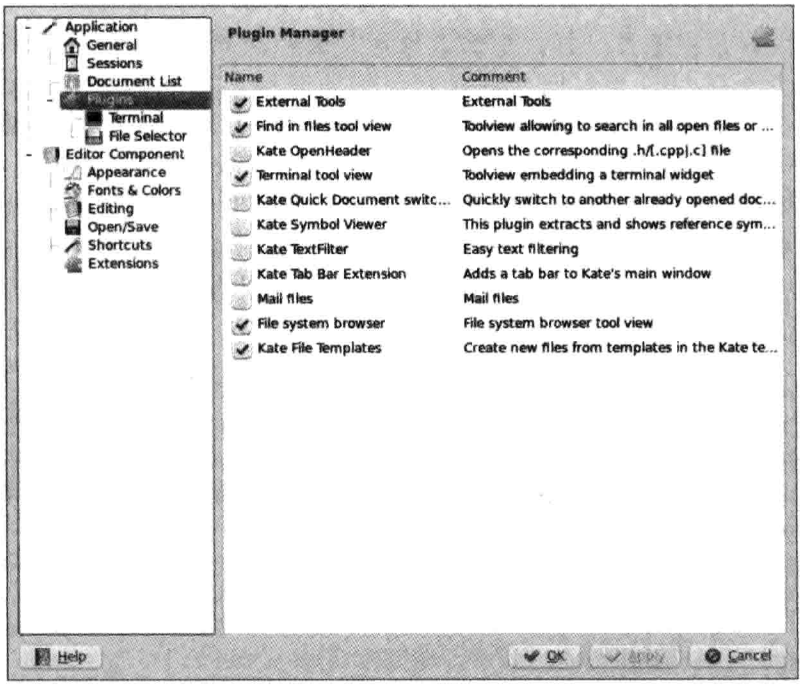


图 6-4 Kate 编辑器的插件管理器

- **启用终端同步(Enable Terminal synchronization)**：在 Plugins 对话框中，选择 Terminal。这里只有一个复选框，当选中时，Kate 将工作目录(如“终端”窗格中显示的) 改变到当前打开文件的目录。从本质上讲，这意味着每当打开一个会话，都将工作目录改变到该会话的文件所在的目录。这是很重要的，在本书其他地方描述 Kate 编辑器的操作时，我将假定这个选项已被选中。

- **启用 Kate 文件模板(Enable Kate File Templates):** 与前一项一样，这也是在 Plugins 对话框中找到的。找到 Kate File Templates 项，并选中该复选框。

其他选项，要么是对显示的微调，要么是与(诸如 C 和 C++之类的)高级语言相关的事情。当你研究 Kate 编辑器时，可能会发现有些设置需要自己定义，但是初学时没有必要这样做。

6.1.4 Kate 会话

如果像大多数程序员一样，那么，你通常会一次处理不止一个项目。我经常同时忙于五六个项目，当时间允许且有灵感的时候，对一个或另一个项目进行“修修补补”。每一个项目都存在于各自的目录中，而且每个项目都有几个文件，对于某些大型项目而言，有时有 10 个甚至 20 个文件。当我决定，已经受够一个项目的“纠缠”，希望转战到另外一个项目时，我不得不关闭 Kate，将工作目录切换到另一个项目的目录，再次运行 Kate 编辑器，然后加载相关的文件到 Kate 编辑器中。这就是我们多年以前的工作方式，相当麻烦，尤其是当编辑器具有类似于 Kate 编辑器的会话特性时。

从根本上来讲，会话(Session)是一个项目的当前状态及其所有各种文件和设置，它保存在磁盘中。当开始一个项目时，可以从磁盘中加载它的会话文件。砰！所有与该项目相关的文件都将出现在侧栏容器中，终端窗格已经跳转到该项目所在的目录。最近打开的那个文件将再次被打开，而且光标仍然停留在你最后一次在此文件中工作的地方。

当关闭会话、更改会话或者关闭 Kate 编辑器时，在引进新的会话或者关闭自身之前，Kate 编辑器将当前的会话状态保存到磁盘中。短短几秒钟的时间，你便可以从一个项目转到另一项目。

处理会话很容易。以下各节描述了会话管理的常见任务。

1. 创建新会话

当启动 Kate 编辑器时，它首先会向你提供一个 Session Chooser (会话选择)对话框(见图 6-1)。此对话框将列出现有的所有会话，并提供一个用于创建新会话的按钮。如果开始一个新项目，单击 New Session(新建会话)按钮。Kate 编辑器将打开一个全新的、空白的会话。

也可以通过选择 Session→New 菜单项随时在 Kate 编辑器内部创建一个新的

会话。这些新会话没有名称，而且不作为会话保存，直到加载一个文件。我建议立即为新会话命名。这是通过选择 Session→Save As 来完成的。一个小对话框将会出现，可以在其中输入会话的名称。输入新会话的名称，然后单击 OK 按钮。

会话名称可以是任何内容。它不一定必须是主.ASM 文件的名称，不必有任何长度限制，也不必符合 Linux 的文件命名约定。但是请注意，直到加载一个文件到编辑器中之后，新会话名称才会在窗口标题栏中出现。

拥有一个已命名会话后，就能够打开该文件，并开始对其进行处理了。可以调整 Kate 编辑器窗口的大小，并且窗口的尺寸将作为会话的一个方面而保存起来，因此，如果源代码文件由于某种原因而具有不同的宽度，可以重新调整 Kate 编辑器的大小，以显示源文件的完整宽度，而且在切换会话时，Kate 编辑器会记住窗口尺寸。

2. 打开现有会话

如同创建新会话一样，打开现有会话可以在启动 Kate 编辑器时完成。Session Chooser 对话框将显示所有 Kate 编辑器的现有会话。高亮显示你想要的会话，并单击 Open Session 按钮。也可从 Kate 编辑器的 Sessions 菜单中打开会话。Sessions 菜单中有两个菜单项允许做这件事情：

- Sessions→Open 弹出一个对话框，可以在其中高亮显示一个会话，然后单击它的 Open 按钮。
- Sessions→Quick Open 弹出一个子菜单，该子菜单列出了 Kate 编辑器的所有会话。选择其中一个，然后单击鼠标左键，即可打开这个会话。

Kate 编辑器没有 Close Session(关闭会话)选项。只能通过加载另一个不同的会话或者创建一个新的会话来关闭一个会话，或者干脆完全关闭 Kate 编辑器。如果想要一个空会话，必须创建一个新会话，记住，Kate 编辑器是不会保存新会话的，除非将其保存到一个特定的名字下。

3. 删除或重命名会话

删除或重命名会话是通过使用 Manage Sessions(管理会话)对话框完成的，可以通过 Kate 编辑器菜单中的 Sessions→Manage 来进行调用。该对话框显示所有已经存在的会话。单击选择一个特定的会话之后，既可以通过单击 Rename 按钮来对其进行重命名，也可以单击 Delete 按钮，将其彻底删除。

请注意，删除一个会话并不会对与该会话相关的文件或目录有任何影响。被删除的只是 Kate 编辑器的会话记录。文件本身不会受到任何影响。

另外一种重命名会话的方法是选择 Sessions→Save As 菜单项，然后把当前的会话用一个新的名称保存。在此之后，从 Manage Sessions 对话框中删除原有的会话。

6.1.5 Kate 编辑器的文件管理

Kate 编辑器可以很方便地加载现有的文件到编辑器窗口，浏览会话工作目录，创建新的文件，重命名文件和移动不必要的文件到回收站。

进行文件管理的主要机制是位于 Kate 编辑器窗口左侧的侧工具栏。撇开其他插件不说，该管理工具栏主要有以下两个功能：

- 当处于 Document(文档)视图时，侧工具栏会显示与本会话相关的文件。可以通过单击文件列表中的一行，将对应的文档加载到编辑器窗口。当文档视图的文件包含未保存的改变时，该文件的入口将显示一个软盘图标。
- 当处于 Filesystem Browser(文件系统浏览器)视图时，侧工具栏显示该会话的工作目录中存在哪些文件。

可以通过位于侧工具栏左边距处的垂直方向的导向图标按钮来选择显示两个视图中的哪一个。图 6-5 显示了 Filesystem Browser 视图下的管理工具栏。两个视图之间的区别稍微有点微妙，但是很值得记住：“文档”视图显示了什么文件是当前 Kate 编辑器会话的一部分。“文件系统浏览器”视图则显示了磁盘上当前会话所描述项目的工作目录下都有哪些文件。从“文档”视图中删除一个文件将会把该文件从会话中删除。从“文件系统浏览器”视图中删除一个文件将会把该文件从磁盘中删除。千万不要把二者混为一谈！

如果显示器上有足够的水平空间，可以在 Filesystem Browser 中显示文件详情。默认情况下，Kate 编辑器只在浏览器中显示文件名。要想显示文件详细信息，右击“文件系统浏览器”的任意位置，从上下文菜单中选择 View→Detailed View 子菜单。详细视图提供了文件的大小、类型以及最后修改的日期和时间。这通常会把管理工具栏的宽度变为原来的三倍，但如果工作目录下有很多文件，这样会带来很大方便。

以下各节简要描述了可以从 Kate 编辑器内部执行的最重要的文件管理任务。

1. Filesystem Browser 导航

在 Filesystem Browser 的顶部，水平排列着 8 个小按钮。这些按钮用于实现

文件夹导航。下面是这些按钮的功能，从左边的按钮开始，依次往右。

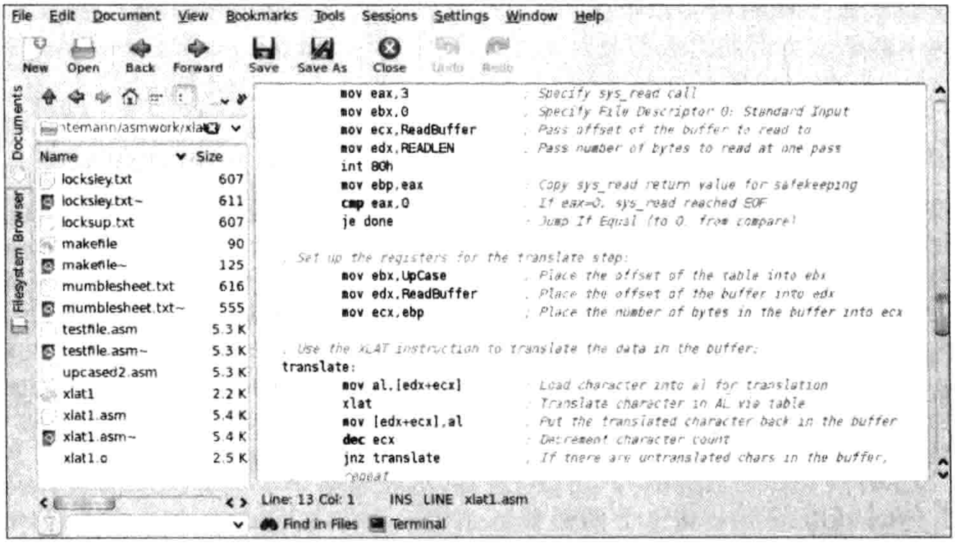


图 6-5　Filesystem Browser 视图

- 蓝色的上箭头：将浏览器转到当前文件夹的父文件夹。当到达根目录时，箭头变为灰色。
- 蓝色的左箭头：将浏览器转到它最后一次所在的文件夹。它与 Web 浏览器和单独的文件管理器中的“返回”按钮采用同样的方式工作。
- 蓝色的右箭头：向前转到一个已经访问的文件夹。它与 Web 浏览器和单独的文件管理器中的“向前”按钮采用同样的方式工作。
- 房子图标：转移到 home 文件夹。
- 接下来的两个图标用于管理 Filesystem Browser 视图的详细程度。左侧图标代表 Short(简短)视图。右侧图标代表 Detailed(详细)视图。
- 金色的星星图标：弹出一个子菜单，用于管理文件夹书签。
- 蓝色的双向箭头：回到当前编辑器中的文件所在的文件夹。

2. 添加一个文件到当前会话

从 Filesystem Browser 视图：单击任意一个文件，该文件都将被加载到编辑器，并添加到会话中。如果单击的是一个二进制文件而不是文本文件，Kate 编辑器将弹出一个对话框，警告你二进制文件无法编辑，如果从 Kate 编辑器保存到

磁盘，可能遭到损坏。(Kate 编辑器是一个文本编辑器而不是 Bless 之类的十六进制编辑器，它不能编辑二进制文件。)除非显式关闭添加到某一会话的文件，否则它将一直是该会话的组成部分。

3. 从当前会话中删除一个文件

从 Document 视图：右击打算从会话中删除的文件，从弹出的快捷菜单中选择 Close 菜单项。该文件将被删除，但是请记住，它不会被真正删除，只是从与当前会话相关的文件列表中删除掉了。

4. 在编辑器中切换会话文件

从 Document 视图：单击想要加载到编辑器中的文件，它将成为被编辑的文件。如果只是在窗口中简单地切换当前文件，先前在编辑窗口对文件所做的更改将不会写入磁盘！当在 Document 视图中单击文件名称时，只是在文件窗口中切换显示的文件。会话中的所有文件都可以处于编辑状态下，只是在视图之间快速跳转。还可以通过主工具栏上的蓝色后退箭头和前进箭头在会话文件之间进行切换。

5. 创建一个全新的文件

单击主工具栏上的 New(新建)按钮，或者从主菜单选择 File→New。一个临时的名为 Untitled 的新文件将出现在文档视图中，并且将成为当前在编辑器窗口中显示的文件。可以立即开始输入文本，但是该文件不会被保存到磁盘，直到你选择或单击 Save As 菜单项，并为这个新文件指定一个名称。快捷键：Ctrl + N。

6. 在磁盘上新建文件夹

从 Filesystem Browser 视图：在文件列表之外的空白处(即不要在文件条目上右击) 右击，并从弹出的快捷菜单中选择“新建文件夹”菜单项。将出现一个对话框，可以在其中为新文件夹输入一个名字，默认情况下该文件夹被创建在当前目录下。

7. 从磁盘删除文件(将文件移动到回收站)

从 Filesystem Browser 视图：高亮显示即将被删除的文件并右击它。然后从弹出的快捷菜单中选择 Move to Trash(移动到回收站)。Kate 编辑器会显示一个确认对话框，以确保你的确想删除该文件。单击 Trash 按钮，该文件将被移至“回收站”文件夹。快捷键：Del。

8. 从磁盘重新加载文件

从主菜单选择 File→Reload。如果修改了原本不打算修改的文件，并希望返回该文件最后一次被保存到磁盘的状态，可以用该命令来方便地实现。该命令将把文件从磁盘加载回编辑器窗口，但是对文件的任何未保存的更改将丢失。快捷键：F5。

9. 保存会话文件中所有尚未保存的更改

从主菜单中选择 File→Save All。如果存在任何新建但尚未保存到磁盘的文件，Kate 编辑器将为其显示一个 Save As 对话框，让你能够命名和保存它。注意，当退出程序时，Kate 编辑器会将所有尚未保存的更改保存到磁盘上，但是如果在工作期间电源发生故障，部分或所有未保存的更改可能会丢失。这就是为什么我在主工具栏上添加一个 Save All 按钮，以及为什么经常在较长的会话中点击它的原因。(稍后我将解释如何添加按钮到工具栏。) 快捷键：Ctrl +L。

10. 打印编辑器窗口中的文件

从主菜单选择 File→Print。将会出现“打印”对话框，在其中可以选择打印到哪个打印机，或对打印作业进行其他配置。快捷键：Ctrl +P。

11. 将文件导出为 HTML

从主菜单中选择 File→Export as HTML。如果需要将某一源代码文件发布到网站上，可以使用这条命令。该命令弹出一个 Save As 对话框，根据你输入的名称创建新的 HTML 文件，该文件包含了编辑器窗口中的当前文件，而且添加了源代码格式。其实现理念是导出的 HTML 文件将会在 Web 浏览器中显示，而且其外观跟该文件在 Kate 编辑器窗口中显示的内容完全一样，包括语法高亮显示和所有的其他格式等。

6.1.6　向工具栏添加菜单项

Kate 编辑器的默认工具栏已经包含了最常用的命令按钮。这个工具栏是可以配置的，然而，我觉得至少有一个按钮应该出现在那里：Save All 按钮。只需点击一次，该命令就会把作为会话一部分而打开的所有文件上的所有未保存更改保存到磁盘上。建议在退出 Kate 编辑器或手动保存对个别文件的更改之前，每隔几分钟单击一下 Save All 按钮，因为可能发生概率极小的断电事故。

下面是完成这项工作的步骤：选择 Settings→Configure Toolbars。Toolbar(工具栏)对话框将出现(见图 6-6)。向下滚动对话框左边的可用行为列表，直到看到 Save All 命令。单击以高亮显示它。然后单击右箭头。Save All 命令将进入对话框中的右侧列表，该列表代表那些按钮将出现在主工具栏上。

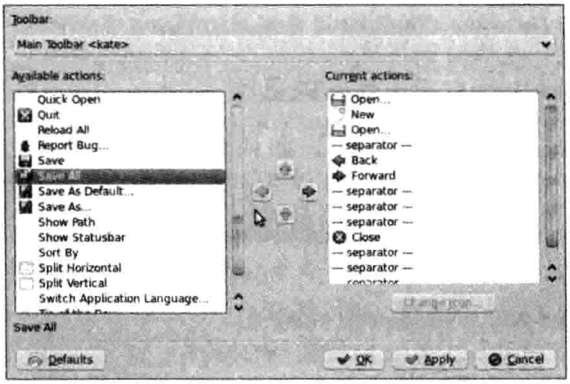


图 6-6　工具栏对话框

默认情况下，Save All 按钮将被添加到工具栏按钮列表的底部。如果希望该按钮出现在 Save 以及 Save As 按钮的上面，单击 Save All 列表项，让它高亮显示，然后单击上箭头，向上移动 Save All 按钮，使其越过那些按钮即可。

在我撰写本书期间使用的 Kate 编辑器版本中，有一个特殊的地方：Save 和 Save As 按钮并不显示在对话框右侧的按钮列表中。为了让 Save As 加入从事“拯救工作”的其他兄弟按钮行列，向上移动它，直到刚好位于两个蓝色箭头下面的分隔线的下方。然后单击 OK 按钮，你会在工具栏中看到 Save As 按钮，而且就在它应该出现的位置上。尽早单击它，而且要经常单击它！

6.1.7　Kate 编辑器的编辑控制

当然，Kate 编辑器最重要的用途是编辑源代码文件。在 Kate 编辑器中花费时间最多的就是编写源代码，因此，尽快记住它的编辑控制是非常有用的。当然，除了实践之外，没有其他的好办法，但是为了开始该项工作，下面的小节将介绍如何在 Kate 编辑器中“走来走去”，以及如何执行基本的编辑任务。

1. 移动光标

在 PC 类型的系统中，基本的光标移动都遵循以下常规做法。

- 左、右箭头键将光标向左或向右移动一个位置。如果光标在某一行的任意一端，光标将移动到预期的相邻行。
- Ctrl +左箭头将光标向左移动一个字；Ctrl +右箭头将光标向右移动一个字；如果是在某一行的结尾，二者皆换行。
- 上下箭头键将光标向上或者向下移动一行。
- Ctrl +向上箭头和 Ctrl +向下箭头实现向上或向下滚动视图而不必移动光标。放开 Ctrl 键，再一次按下箭头键，将返回到当前光标的所在位置(这可以让你向上或者向下查询文件一段时间而不会失去光标的所在位置，也就是你在当前文本中正工作的地方)。
- Home 键将光标移到当前行的左端。但是，如果文本是缩进的，光标将移动到最左边的第一个可见字符处。
- Ctrl+Home 将光标移动到文档的开始位置。
- End 将光标移动到当前行的右侧末尾位置。
- Ctrl+End 将光标移动到文档的末尾。
- Page Up 键和 Page Down 键分别将光标向上或者向下移动一定数量的行，该行数由在当前大小的窗口中所显示的行数决定(例如，如果屏幕大小设置为显示 30 行，Page Up 键 和 Page Down 键将光标移动 30 行)。
- 可以通过 Ctrl+G 键移将光标移到文件中某个特定行。编辑器窗口的底部将出现一个页码域。在页码域中输入目标行的行号，然后按 Enter 键即可移到该行。

2. 书签

Kate 编辑器允许在文件中设置书签。这是一条对文件要点提供快速导航的方便途径，尤其是当文件多达数百甚至上千行时。例如，我一般在.DATA 段、.BSS 段和.TEXT 段的起始行设置标号(详见后文)，这样，如果必须增加或编辑预定义的数据项，我可以迅速返回至.DATA 段，而不必来回滚动代码来寻找它。

设置书签的最简单方法是单击打算设置书签的行的左侧图标边框。这就要求该图标边框可见，如果不可见，可以按 F6 键来显示它。书签设置完毕后，在设置了书签的那一行的图标边框中，将显示一个小小的金色星形图标。

还可以通过另外一种方法设置书签：将光标置于想要放置书签的行上，然后右击该行，从弹出的快捷菜单中选择 Bookmarks→Set Bookmark。任何设置了书签的行都将通过背景颜色突出显示。

切换书签：在图标边框内部单击(或按下快捷键 Ctrl+B)会设置一个书签，单

击书签上的金色星形图标，或者再次按 Ctrl +B 键可将书签清除。

可以通过选择 Bookmarks→Clear All Bookmarks 一次性清除文件内部的所有书签。

若要移到某个书签，打开 Sessions 菜单，做以下两件事情之一。

- 在菜单底部列表中找到想要的书签并单击。书签上“贴”了一个标签，上面存放该书签所在行的行号以及源代码文本。
- 选择 Previous 或者 Next，从当前光标位置移至上一个书签或者下一个书签。

3. 选择文本

像在所有文本编辑器中一样，在 Kate 编辑器中可以选择文本以实现删除，或者通过单击拖放来移动文本，或者将文本置于剪贴板中。可以通过以下几种方法来实现文本选择。

- 将光标移至某个词上，双击该词即可选择它。
- 将光标放在某一行的任意位置，三击鼠标可选中整行。
- 将光标放在想开始选择的地方，然后按下鼠标左键并拖动鼠标到所需选择的另一端。有趣的是，以这种方式选择文本会自动把选择的内容复制到一个单独的剪贴板中，而且可以粘贴到文档中，只需单击鼠标中键即可(如果有这个键的话)。按 Ctrl + V 或者选择 Edit→Paste 都做不到这一点。以前通过 Ctrl + C 快捷键选择的文本或者通过 Edit→Copy 菜单选择的文本，将依然存放到传统的剪贴板中。注意，在大多数现代鼠标中，鼠标滚轮作为中键使用，也可以像左右键一样按下。通过单击并拖动而选中的文字可使用传统的编辑命令操作，如通过 Ctrl +C 键、Ctrl + X 键和 Ctrl +V 键。
- 通过选择 Edit→Block Selection Mode(快捷键：Ctrl+Shift+B)，可以选择矩形文本区域，而不用“尊重”行的长度。当 Block Selection Mode(块选择模式)有效时，按下鼠标左键并拖动鼠标将选择一个区域，起点在一个拐角，结束点(你松开鼠标左键的地方)在对面的拐角。
- 文本可以通过按住 Shift 键，同时按住各导航键从键盘进行选择。光标将正常移动，原来的光标位置和新的光标位置之间的所有文本将被选中。

可以把选择的文本拖放到文件中的某个新位置，常用的方法是：在移动鼠标指针的同时，按住鼠标左键不放。

4. 查找文本

Kate 编辑器的文本搜索功能通过一个名为搜索栏(search bar)的内置对话框来实现，当需要时，可以让该对话框出现在编辑器窗口的底部。有两种形式的搜索栏：一个用于简单的增量搜索，另一个用于管理搜索和替换操作。

选择 Edit→Find 或按 CTRL+F 键，可弹出增量搜索栏。默认情况下，出现的这个搜索栏管理一个简单的、增量的搜索机制。在“查找”域中输入文本，在输入文字期间，Kate 编辑器将查找与正在敲入的文本匹配的第一个增量文本(见图 6-7)。

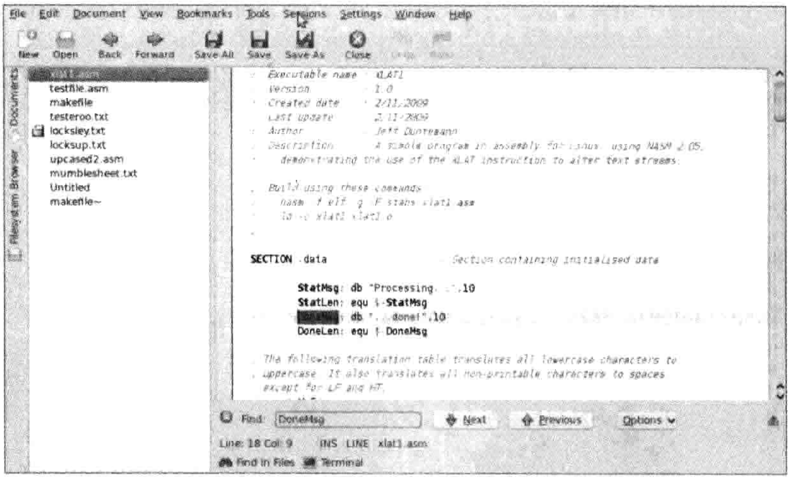


图 6-7　增量搜索栏

在图 6-7 中，我输入了 DoneMsg，当我仍然在打字的时候，Kate 编辑器找到并高亮显示该文件中第一个与该文本匹配的实例。那个高亮显示的文本已经被选中，可以通过拖动来移动它，或通过使用传统的编辑命令剪切或复制它。Next 和 Previous 按钮分别用于搜索当前光标位置之后或之前的相同文本。

位于搜索栏右端的 Options(选项)菜单允许对搜索过程进行一些优化。如果选择 Options→Highlight All(高亮显示所有内容)，Kate 编辑器将选择下一个与搜索文本相同的实例，但它会以不同的颜色高亮显示文件中所有与搜索关键字相同的实例。如果选择 Options→Match Case(匹配大小写)，搜索将区分大小写；否则，文本中的大小写差异将被忽略。

可以通过单击增量搜索栏左上角的红色关闭按钮来隐藏它。

5. 使用查找和替换

搜索和替换是由第二个搜索栏处理的，也可以通过 Edit→Replace 菜单，或者 Ctrl +R 快捷键来调用它(见图 6-8)。像简单搜索那样在“查找”域输入文本之后，Kate 编辑器将执行增量搜索，查找并选择文件中与搜索关键字相同的第一个实例。如果在“替换”域中输入了一个替换文本，那么当单击“替换”按钮时，Kate 编辑器将用“替换”域中的文本来代替文件中与“查找”域中文本相同的文本。或者，如果单击 Replace All(全部替换)按钮，Kate 编辑器将对整个文件执行搜索和替换操作，用“替换”域中的文本替换文件中与“查找”域中文本相同的所有实例。

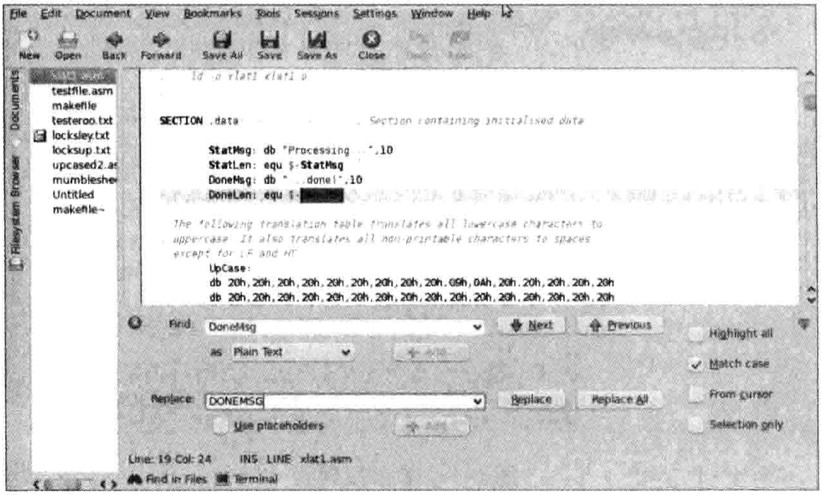


图 6-8　查找和替换栏

管理搜索的几个选项都显示为按钮(在右侧)，而不是下拉菜单项。这些选项工作起来与它们在增量搜索栏中一样。还有一个附加选项：如果选中了 Selection Only，搜索和替换操作将只发生在已选定文本的内部。

搜索和替换栏都可以通过单击其左上角处的红色关闭按钮来隐藏。

6.1.8　编程期间使用 Kate 编辑器

至少对于本书提供的程序而言，Kate 编辑器将成为我们编辑、编译、连接、测试和调试代码的“工作台”。换句话说，从应用程序菜单或桌面或面板启动

Kate 编辑器，然后所有工作都在 Kate 编辑器内完成。

这里的“内”有一个有趣的创新，即 Kate 编辑器拥有自己的内置 Linux 终端窗口，该终端窗口能够让我们从 Kate 编辑器内部启动其他工具：具体来讲，包括 Make 实用工具(下一节将进一步介绍)以及任何打算使用的调试器。所有可以在 Linux 控制台中做的事情，都可以在 Kate 编辑器的终端窗口中完成，默认情况下，它是连接到 Linux 控制台的。

虽然可以从终端窗口内部启动 Kate 编辑器，但我并不推荐这样做。因为当从终端启动时，Kate 编辑器会连续不断地向窗口发布与自己内部机制相关的状态信息。这样做主要是为了方便从事 Kate 编辑器相关工作的程序员，但是对于 Kate 编辑器的用户而言，意义不大甚至毫无意义，反而容易分心。更好的做法是在桌面或面板上创建一个启动图标，然后通过这些图标来启动 Kate 编辑器。

1. 创建和使用项目目录

正如本节前面所述，汇编语言项目可以很好地映射到 Kate 编辑器关于会话的理念。项目应该位于汇编语言工作目录下的自己的目录中，而汇编语言工作目录应该驻留在 Ubuntu Home 目录下。所有与项目相关的文件(除了项目之间共享的库文件之外)应该保存在该项目的目录下。在本书中，我们将整个汇编语言工作目录称作“asmwork”。

我已为本书创建的示例代码是一个 ZIP 存档文件，并且当将其解压缩到硬盘驱动器中你自己的汇编工作目录下时，将为每一个示例程序创建一个项目目录。如果是第一次从 Kate 编辑器中打开这些“已经做好的”项目目录，应遵循如下步骤。

(1) 启动 Kate 编辑器。

(2) 当 Session Chooser 对话框出现时，单击 New Session 按钮。

(3) 单击主工具栏上的 Open 按钮，或者选择 File→Open 菜单项。使用出现的 Open File 对话框，从 Home 目录开始，导航到想要的项目目录。

(4) 当显示项目目录中的文件时，打开源代码文件和生成文件。可以在一次操作中完成这项任务：按住 Ctrl 键的同时，单击希望打开的所有文件。所有选定的文件将被高亮显示。然后单击对话框里的 Open 按钮，将它们全部打开。

(5) 一旦所有打开的文件都显示在 Kate 编辑器侧工具栏的文档视图中，使用一个描述性名称保存这个新的会话。选择 Sessions→Save As 菜单项，在 Session Name 对话框中输入新会话的名称。单击 OK 按钮，此时在项目

目录下就有了一个新的 Kate 会话。

如果想创建一个全新的项目，而且不存在该项目的相关目录时，应遵循如下步骤。

(1) 启动 Kate 编辑器。

(2) 当 Session Chooser 对话框出现时，单击 New Session 按钮。

(3) 单击 Filesystem Browser 按钮以显示“文件系统浏览器”视图。

(4) 使用 Filesystem Browser 导航到汇编语言工作目录(例如：asmwork)。

(5) 在“文件系统浏览器”视图中，右击任意位置，并从弹出的快捷菜单中选择 New Folder 菜单项。为新的目录键入一个新的名称之后，单击 OK 按钮。

(6) 采用步骤(5)所述的方法，以这个新的名称保存新建会话。

此时，可以开始为新项目输入源代码文本了。不要忘记定期单击 Save All 按钮来定期保存它。

Kate 编辑器的终端窗口默认情况下并不显示。可以通过单击 Kate 编辑器窗口底部边缘的 Terminal 图标来显示它。注意，只有从 Kate 编辑器中至少保存一个文档到刚刚创建的全新项目目录之后，终端才会显示那个目录。Kate 编辑器将终端同步到显示在编辑器窗口中的那个文档的位置，并且让终端窗口一直保持该位置，直到编辑器窗口中出现一个已被保存到目录树中某个特定位置的文件。

一旦有一个项目目录，并且已开始从 Kate 编辑器中保存文件到该目录，可以按以下工作流程进行操作。

(1) 在 Kate 编辑器的编辑窗口中输入并且修改源代码文件和生成文件。把所有的修改保存到磁盘。

(2) 通过在终端窗口中单击项目使其获得焦点，然后在终端命令行中输入“**make**”命令来生成它。假设已经为该项目创建了一个生成文件，Make 实用工具将启动 NASM 和连接器，并显示发生在生成过程中的每一个错误。(本章后面部分将进一步介绍 Make。)

(3) 测试可执行程序，确保终端窗口具有焦点，然后输入 **./myprog**，其中“myprog”是要测试的可执行文件的名称。

(4) 如果想观察运行在调试器下的程序，确保终端窗口具有焦点，然后输入调试器的名称，后面跟可执行文件的名字，例如：insight myprog 。调试器将出现，同时你的程序已被读取和加载完毕，留待你进行观察。完成调试之后，退出调试器，返回 Kate 编辑器。

(5) 单击编辑器窗口以使其获取焦点，借此返回编辑器窗口，继续处理源文件。

2. 焦点

初次学习使用 Kate 编辑器时，“焦点在哪里”这个问题可能会给你来个“下马威”。在任何现代窗口环境中(如 GNOME 或 KDE)，焦点本质上是指接收按键输入内容的屏幕组成部分。大多数时候，是在 Kate 编辑器的编辑窗口中输入内容，因为当你编辑文件时，编辑器窗口具有焦点。如果稀里糊涂地忘记将焦点移到终端，你最终很可能将 make 或其他 Linux 命令输入到自己的源代码文件中，如果不经意地这样做了(因为在软件创作热情的鼓动之下，很容易疏忽)，那么，在下一次尝试编译该项目时，你很可能会在错误消息中发现这些被 NASM 特别强调的文本。

终端窗口是否具有焦点非常容易分辨：如果终端的块光标是一个空心方块，说明焦点在其他地方；如果终端的块光标是一个实心方块，说明终端具有焦点。

6.2 Linux 和终端

那时，使用 Unix 的人们讨厌接收终端，但是在它创建之后，Unix 才真正名副其实地成为大型机(如 IBM)操作系统，并且能够通过分时系统支持多个用户同时使用。每个用户通过独立的终端(尤其是 DEC 公司的 VT 系列)与中央计算机进行通信。

这些终端都不显示自从 1995 年以来已经成为主流的图形化桌面。它们是纯文字设备，通常呈现 25 行，每行 80 个字符，没有图标或者窗口。有些应用使用全屏显示，能够呈现数字化菜单并为输入数据项而准备文本域。大部分 Unix 软件工具，特别是那些程序员们使用的工具，都是从命令行来进行控制的，并且发回从底往上滚动的输出。

Linux 以同样的方式工作。简单来说，Linux 就是 Unix。 Linux 不使用外部“哑”终端，例如 20 世纪 70 年代的 DEC VT100，但是 DEC 式的面向终端的软件机制仍然存在于 Linux 内部，并且仍在以仿真终端的形式运转。

6.2.1 Linux 控制台

有很多 Linux 终端仿真程序和其他的 Unix 实现，例如 BSD。Ubuntu 配有一

个名为 GNOME 的终端，也可以下载并从 Applications→Add/Remove 菜单项安装许多其他类型的终端。本书讨论中使用的且一般情况下推荐的终端是 Konsole。如果还没有，一定要安装它。

在 Linux 下打开一个终端仿真程序时，将会得到一个含有闪烁光标的文本命令行，就像老式的 DOS 命令行或者 Windows 下的命令提示符实用工具一样。从 Unix 的第一代起，终端程序就尽力像那些老式的 DEC CRT 系列终端一样工作。默认情况下，终端仿真程序使用 PC 键盘和显示器进行输入和输出。它连接到一个名为 dev/console 的特殊 Linux 设备，这是一个预定义设备，可提供与 Linux 系统本身的通信。

记住这一点是非常有用的，终端程序只是一个程序，可以在 Linux 机器上安装各种不同类型的终端程序，每个终端程序可在同一时间内运行多个实例。但是，只有一个 Linux 控制台，我指的是名为 dev/console 的设备，它传送命令到 Linux 系统，并返回系统的响应。默认情况下，当启动的时候，终端仿真程序连接到 dev/console。如果愿意，可以使用 Linux 的终端仿真程序通过网络连接到其他设备，至于它是如何工作的，以及应该如何实现，已经超出了本书讨论的范畴。

与 Linux 程序通信的最简单方法是通过像 Konsole 程序一样的终端仿真程序，这是本书中将谈及的方法。终端仿真程序的一种替代方案是为某种类型的窗口系统编写程序。描述 Linux 桌面管理器以及在底层操纵它们的 X Window 系统将花费整本(或几本)书的篇幅，并且涉及与汇编语言无关的复杂层面。因此，本书的示例程序将严格从终端仿真程序命令行进行操作。

6.2.2　Konsole 中的字符编码

在终端仿真程序中没有多少配置，至少在你刚刚涉足汇编语言的时候是这样。有一点倒是的确对本书的示例程序非常重要：字符编码(character encoding)。终端仿真程序必须将字符放入它的窗口，其中一个可配置的选项与哪个字形对应哪个 8 位字符编码有关。

请务必注意，这与字体没有任何直接的关系。字形是一个特定的识别符号，像字母“A”或“@”符号。如何以图形的方式呈现符号取决于使用的字体。一个特定标志呈现在不同的字体中，可能是胖的或瘦的，或下标形式的，或各种各样类型的花式。可以在任意数量的字体中显示“A”，但是假设字体没有装饰过度 (并且存在这样的字体)，你仍然可以看出某个特殊的字形是 “A”。

字符编码把一个数值映射到特定的标志符号。在我们熟悉的西方 ASCII 标准

中，数字 65 与我们能够识别的作为大写字母“A”的字形相关。在一个不同的字符编码体系中，编码被创建用于显示一个完全不同的非罗马字母(如希伯来语、阿拉伯语或泰国语)，数字 65 可能与一个完全不同的字形相关联。

本书采用西方人和大多数讲英语的读者使用的罗马字母表编写，因此，终端仿真程序的默认字母表的字形做得很好。然而，ASCII 字符集实际上只是从字符 0 到字符 127。8 位二进制数能够表示的最大值是 256，所以有另外 128 个“高端”字符超出了 ASCII 顶端的标准。这 128 个高端字符对应哪个字形，没有硬性的标准，当然也不像 128 个较低字符的 ASCII 标准那么强硬。高端的 128 个字符的不同字符编码方案包括很多种不同的字形，其中大多数是带修饰符(元音变音、抑扬符号、鼻音化符号、重音符号等)的罗马字符、主要的希腊字母以及来自数学和逻辑的符号。

当 IBM 于 1981 年发布其最原始的 PC 时，它包含了早期为其大型机终端所创造的字形，能够在纯文本的、无法显示像素图形的终端屏幕上呈现方块。事实证明，这些字形在界定字符代替者以及其他事情方面是非常有用的。PC 的基于 ROM 的字符集最终被称为代码页 437(CP437)，其中包含很多其他符号，如 4 个纸牌花色。

类似的字符编码方案后来被用于 IBM 的 Unix 实现——AIX 上，它被称为 IBM-850。IBM-850 包含 CP437 的方块字符的子集，再加上很多带有修饰符的罗马字母字符，从而能够正确呈现除了英语之外的其他西方语言文字。

默认情况下，Linux 终端仿真程序既不采用 CP437 编码方案编码，也不采用 IBM-850 方案编码(因此，不会有方块字符)。IBM-850 编码方案是可以使用的，但是必须从菜单中选择它。在本书的后面部分中，我们需要用到那些方块字符，所以现在是一个描述如何选择它们的好时机。(顺便说一句，在撰写本书时，我还从来没有见过 Linux 终端仿程序能够显示 IBM 的原始的 CP437 字符集，如果你知道一个，千万别忘了交流一下。)

启动 Konsole，打开 Settings→Manage Profiles(管理配置文件)菜单项。Konsole 带有一个名为 Shell 的配置文件。在 Manage Profiles 对话框中，选择 New Profile(新建配置文件)，并给它取一个名字，如“Shell Box”。在 Edit Profile(编辑配置文件)对话框中，选择 Advanced(高级)选项卡，并在窗格的底部查找 Encoding(编码)下拉菜单。单击 Select(选择)，将出现一个列表，让鼠标悬停在 Western European(西欧)菜单项上，直到出现编码列表。选择 IBM-850，然后单击 OK 按钮(见图 6-9)。

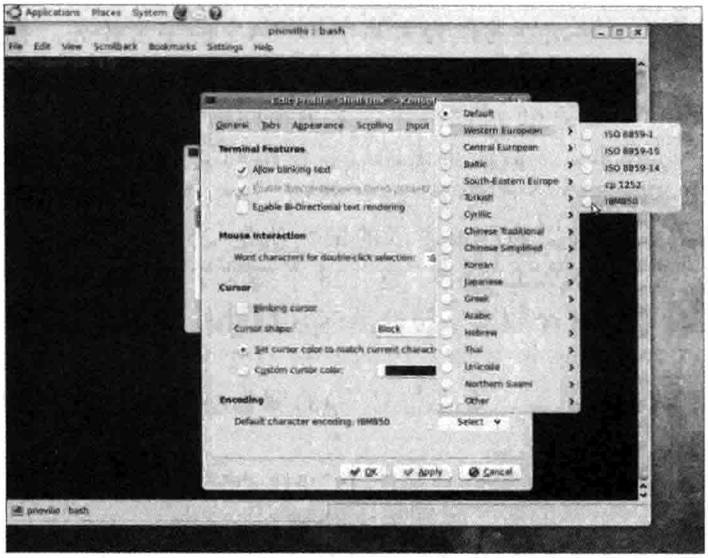


图 6-9 将 Konsole 的字符编码改为 IBM-850

当 Shell Box 配置文件生效时，就可以在程序中使用 IBM 的方块字符了。

6.2.3 三个标准 Unix 文件

人们将计算机描述为移动数据的机器，这是一种不错的看法。不过，要想通过终端仿真程序来控制程序的输入和输出，最好的方法是理解 Unix 的一个根本设计原则：一切都是文件。文件可以是磁盘上的数据集合，这一点我已经在第 5 章进行了比较详细的讲解；但是用更一般的术语来说，文件是数据所采用的路径的端点。当将数据写入一个文件时，你在沿着一条路径发送数据到一个端点。当从文件中读取数据时，你在从一个端点接收数据。数据在文件之间采用的路径可能完全位于一台计算机的内部，或者位于某种类型的网络计算机之间。数据可以被沿着路径处理或修改，或者它只是从一个端点移动到另一个端点，未作任何修改，无论怎样均可。一切都是文件，Unix 的内部文件机制一视同仁地对待所有文件。

“一切都是文件”不只适用于磁盘上的数据集合。键盘是一个文件：它是一个生成数据并将其发送到某个地方的端点。显示器是一个文件：它是一个从某处接收数据并将其放在一个你可以看到的地方的端点。Unix 下的文件不一定必须为文本文件。二进制文件 (如 NASM 和连接器所创建的可执行文件) 也以相同的方式处理。

表 6-1 列出了 Unix 定义的三个标准文件。当程序正在运行期间，这些文件对程序一直是打开的。

表 6-1 三个标准的 Unix 文件

文件	C 标识符	文件描述符	默认值
Standard Input	stdin	0	键盘
Standard Output	stdout	1	显示器
Standard Error	stderr	2	显示器

在底层，操作系统通过文件描述符来知道一个文件，文件描述符只是一个数字。前三个这样的数字属于三个标准文件。在某一程序内部打开一个现有文件或者创建一个新文件时，Linux 会返回一个针对该文件的描述符值。为了操作该文件，需要调用操作系统，向它传递你想处理的文件的文件描述符。表 6-1 还提供了传统的标识符，标准文件可以通过它们被 C 语言识别。例如，当人们谈论“标准输出”时，他们是在谈论文件描述符 1。

如果回过头来再看一遍程序清单 5-1(这是在第 5 章中介绍汇编语言开发过程时提供的简短示例)，你会看到这样一行：

```
mov ebx,1   ; 指定文件描述符 1：标准输出
```

当我们向显示器发出小广告语“Eat at Joe's!”时，我们实际上是将其写入文件描述符 1，即标准输出。通过将 1 改为 2，我们也能够将该广告语发送到标准错误。它在显示器上显示时没有任何不同。在数据处理的各个方面，标准错误与标准输出是完全相同的。按照惯例，NASM 之类的程序将错误信息发送到标准错误，但是，写入标准错误的文本并不标记为“错误消息”或者显示为不同的颜色或字符集。标准错误和标准输出的存在，使得我们可以把程序的输出与程序的错误以及其他与程序功能和实现机制相关的信息区分开来。

了解了 Unix 后裔系统中最有用的基本机制之一——I/O 重定向之后，上述讲解会更有意义。

6.2.4 I/O 重定向

默认情况下，标准输出被发送到显示器(通常是一个终端仿真程序窗口)，但那只是默认情况。你还可以更改来自标准输出的数据流的端点。来自标准输出的数据可以被发送到磁盘上的文件中。文件也是一个文件，Linux 以相同的方式处

理文件之间的数据流动，因此，改变端点也不是什么大的特技。来自标准输出的数据可以被发送到一个现有文件，也可以被发送到程序运行时新建的文件。

默认情况下，程序的输入来自键盘，但是键盘发送的所有内容都是文本。这些文本也可以来自另外一个文本文件。切换程序的数据源并不比切换程序输出的目标困难。完成此类事情的机制称为 I/O 重定向，我们将在本书的很多示例程序中使用到它。

也许你在之前的 Linux 工作中已经使用过 I/O 重定向技术，尽管还不知道它的名字。Linux 的所有基本外壳命令都将其输出发送到标准输出。例如，ls 命令将一个工作目录的内容列表发送到标准输出。可以通过将 ls 发出的文本重定向到一个 Linux 磁盘文件来捕捉该列表。为此，在命令行中输入以下命令：

```
ls > dircontents.txt
```

如果 dircontents.txt 文件不存在，将创建该文件，并且 ls 发出的文本被存放在 dircontents.txt 中。然后，你可以打印这个文件或者将其加载到一个文本编辑器中。

“>”符号是两个重定向操作符之一。“<”符号以相反的方式工作，它能够使原本来自键盘的标准输入重定向为来自另一个文件，而该文件通常是存储在磁盘上的文本文件。它可用于将键盘命令提交到程序，但是更主要的用途是提供程序即将处理的原始材料。

假设你要编写一个程序，该程序可实现将文件中所有的小写文本强制转换为大写字符。(这是一种奇妙的逆向操作，因为大写字符会让一些使用 Unix 的人变成半个傻瓜。)你可以将该程序编写为从标准输入中获取文本，并且将其发送到标准输出。从编程的角度来看，这是很容易做到的。事实上，随着本书的深入，我们还可以对其进行更深一步的处理。

可以从键盘输入一行文本来测试这个程序：

```
i want live things in their pride to remain.
```

该程序将处理上面这行文本，并将处理后的文本发送到标准输出，在那里它将被提交给终端仿真程序进行显示：

```
I WANT LIVE THINGS IN THEIR PRIDE TO REMAIN.
```

这个测试是成功的：它看起来像在程序内部自然完成。下一步是在一些真正的文件中测试大写转换。你根本不需要修改这个大写转换程序。只需在 Shell 提示符下键入如下内容即可：

```
uppercaser < santafetrail.txt > vachelshouting.txt
```

通过 I/O 重定向的魔力，该程序将从名为 santafetrail.txt 的磁盘文件中读取所有的文本，并强制将所有小写字母转换为大写字母，然后再将大写文本写入磁盘文件 vachelshouting.txt 中。

我们可以将重定向操作符看作指向数据移动方向的箭头。数据正被从文件 santafetrail.txt 中取出，并送往 uppercaser 程序，因此，符号“<”从输入文件指向它要保存到的程序。该大写转换程序将发送数据到输出文件 vachelshouting.txt 中，因此重定向操作符不是指向程序的名称，而是指向输出文件的名称。

从某一高度来看，其中发生的事情看起来像图 6-10 中所画的一样。I/O 重定向充当一种类型的数据交换机，把数据流从标准文件转向你自己选择的已命名源文件和目标文件。

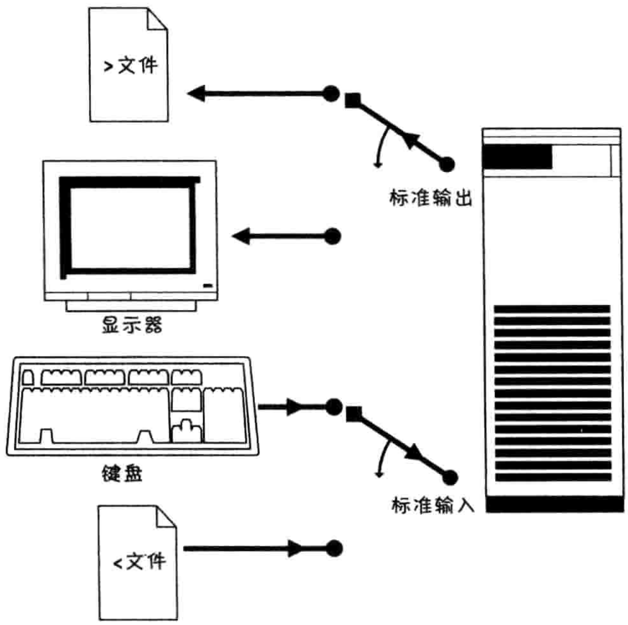


图 6-10　I/O 重定向

6.2.5　简易文本过滤器

我们实际上以后要创建一个名为 uppercaser 的小程序，它要做的事情是：从

一个文本文件中读取文本，处理该文本，然后将处理后的文本写入输出文件。在程序中，我们将从标准输入读取数据，并将处理后的数据写入标准输出。因此该程序没有必要提示用户输入和输出文件名、创建输出文件等。Linux 将为我们完成一切事情，这使得编程任务变得更加容易。

在更大的 Unix 世界中，以这种方式工作的程序代表着一个标准机制，该机制叫过滤器。你已经和其中的两个打过照面了。NASM 汇编编译器本身就是一个过滤器：它读入满是汇编语言源代码的文本文件，对其进行处理，然后写出一个全是目标代码和符号信息的二进制文件。Linux 的连接器读取一个或多个全是目标代码和符号信息的文件，并写出一个可执行程序文件。NASM 和连接器不止能够处理简单的文本，但仅是处理文本这一项就足够了。文件也是一个文件，Linux 采用的文件操作机制在文本文件和二进制文件之间没有任何区别。

过滤程序并不总是使用 I/O 重定向来定位它们的输入和输出。NASM 和大多数连接器选择在命令行中关闭它们的源文件和目标文件名称，这是一个有用的技巧，我将在本书的后面部分加以讨论。I/O 重定向仍然会使简单文本过滤程序的编写变得更加容易。

一旦掌握了过滤器程序的工作原理，就会明白为什么会有标准错误文件以及它们的作用。过滤器程序把输入数据处理成输出数据。在这个过程中，它可能需要发布一些错误信息，或者只是向我们确认一下，它仍然在缓慢而坚定地前进，没有陷入死循环。为此，我们需要一个独立于程序输入输出的沟通渠道。标准错误就提供了这样一种沟通渠道。程序可以通过将这些消息写到标准错误文件来发布文本状态和错误信息到终端显示仿真程序，所有这些都发生在它正在工作、标准输出文件正忙着将程序输出到磁盘期间。

标准错误可以像标准输出一样被重定向。例如，如果想捕捉程序的状态和/或错误消息到一个名为 joblog.txt 的磁盘文件，应该采用以下方式从终端命令行启动该程序：

```
uppercaser < santafetrail.txt > vachelshouting.txt 2> joblog.txt
```

这里，2>操作符指定文件描述符 2(前文讲过，文件描述符 2 代表标准错误)正在被重定向到 joblog.txt。

如果想将(来自任意数据源的)输出重定向到现有的磁盘文件，重定向将用新的数据代替文件中原来可能存在的一切内容，这些旧的数据将被覆盖并且丢掉。如果将重定向数据追加到一个已经包含数据的现有文件的结尾，必须用追加操作符“>>”来替换“>”。

6.2.6　使用转义序列进行终端控制

默认情况下，到终端仿真程序窗口的输出从最底行的左端进入，每在底部增加一个新行，以前显示的行都会往上滚动一行。这绝对是可用的，但是它不够漂亮，因此实话实说，它当然没有资格称为一个“用户界面”。在很久以前，有很多为 Unix 操作系统编写的“全屏”应用程序，它们在全屏范围内显示数据输入域和提示信息。彩色显示终端出现后，文字可以显示为不同的颜色，而其背景也可以设置为白色或其他颜色以便与文本形成鲜明对比。

这是如何实现的呢？老式的 DEC VT 终端，如 VT100，可以通过在标准输出或者标准错误发送到终端的数据流中嵌入一系列字符来进行控制。这些字符序列被称为转义序列(escape sequence)，因为它们是那些即将发送到终端进行显示的普通数据流中的“逃兵”(尽管是临时的)。

VT 终端观察其正在显示的数据流，并挑出具有单独释义的转义字符序列。例如，某个转义序列将被解释为一个命令，用来实现清屏操作；另一个转义序列将被解释为一个命令，用来在距离屏幕顶部 5 行，距离屏幕左边距 30 个字符的位置显示下一个字符。有几十个这种可识别的转义序列，它们让当时相对原始的文本终端能够向用户显示出格式整齐的文本，一次显示满满一屏。

Linux 终端仿真程序(如 Konsole)看起来像一种老式 DEC 终端，至少就它在我们 21 世纪液晶电脑显示器上显示数据方面而言。发送字符序列“Eat at Joe’s!”到 Konsole，它会“乖乖地”把该内容显示到它的窗口中，就像老式的 VT100 一样。程序清单 5-1 就是这样的程序。然而，Konsole 观察我们发送给它的字符流，并且也能识别出这些转义序列。Konsole 关键警惕一个通常情况下并不显示的特殊字符：ESC，与之对应的 ASCII 码为 27，或 01Bh。当 Konsole 在其显示的文本流中看到一个 ESC 字符时，它会非常仔细地查看后面几个字符。如果 ESC 字符后面的前三个字符是[2J，那么 Konsole 确认它为一个转义序列，该序列命令 Konsole 执行清屏操作。但是，如果 ESC 键后面的 4 个字符是[11H，那么 Konsole 看到的是另一个转义序列，该序列命令它将光标移动到显示器左上角的 home 位置。

毫不夸张地讲，有几十个不同的转义序列，它们都代表一些操作命令，如四处移动光标，改变字符的前景色和背景色，切换字体或字符编码，擦除线条或者部分线条或者整个屏幕的一部分，等等。通过向标准输出发送精心设计的转义字符序列，终端窗口中运行的程序能够完全控制整个显示器中的显示。(我们将在

稍后进行一些相关操作，现在只需警告一下，整个实现过程并不像听起来那么简单。）在图形用户界面(GUI)应用程序时代到来之前，Unix 正是通过发送转义序列到终端(或终端仿真程序)来实现编程显示的。

6.2.7　为什么不用汇编语言编写 GUI 应用程序呢

这引发了一个有趣的问题。到目前为止，这本书已经印刷了超过 20 年，我收到了关于这个问题的大量邮件。头号问题(在不可避免的“你能帮我做我的 CS102 项目吗？”之类的问题之后)是这样的：我该如何编写 GUI 应用程序？我的大多数“记者”指的是 Windows 应用程序，而且，到处都有人询问为 GNOME、KDE 或 MOTIF 编写应用程序的相关问题。我已经吸取了多年以前的教训，绝不回答说，“为什么你想这样做？”，而是诚实地用真相来回答：因为此类项目需要相当多的研究和大量的努力，但却回报甚微。

相反，如果你确实学会了编写 Windows 或 Linux GUI 应用程序，你就懂得了这些操作系统的用户界面机制的工作原理，如果有时间和精力，研究此类项目当然是有价值的。

问题是，在介入此类项目时有一个巨大的障碍。在能够使用汇编语言编写第一个 GUI 应用程序之前，你必须要了解它的“所有”工作原理，并且为了迎接这一特殊的挑战，有很多很多的“所有”。GUI 应用程序需要处理操作系统发射的、用来表明按下了哪个键或者点击了哪个鼠标按钮的*信号*(在 Windows 中，被称为*事件*)。GUI 应用程序不得不管理一个由按钮、菜单和填写域构成的庞大而复杂的“控件集”，以及数量多得简直令人难以置信的应用编程接口(API)调用。当一个应用程序的部分屏幕显示区域“变脏”(被某些其他东西覆盖或被应用程序更新)时，或者当用户重新调整应用程序的一个或多个窗口的大小时，需要进行管理内存并对其进行重绘。

Windows GUI 编程的内部机制是我见过的最丑陋的事情之一。(Linux 也同样很复杂，虽然不是那么难看。)幸运的是，这是一种非常标准化的丑陋，很容易封装到代码库中，而代码库是不会因应用程序的不同而有大幅改变的。这就是为什么图形化 IDE 和非常高级的编程语言产品如此流行的原因：它们把大多数与操作系统 GUI 机制的接口的丑陋隐藏在一套工作于某个应用程序框架下的标准类库后面。你可以使用 Delphi 或 Visual Basic(用于 Windows)或者使用 Lazarus 或 Gambas(用于 Linux)编写非常好的应用程序，忽略底层工作原理即可。如果想使用汇编语言，在开始工作之前必须知道一切机制。

这意味着你必须从其他某些地方开始。如果你真心希望为某种 Linux 桌面管理器编写基于汇编语言的 GUI 应用程序，可以通过以下步骤来接近这一目标。

(1) 采用一种强大的本地编码高级语言(如 Pascal、C 或者 C++)研究 Linux 编程。而这里，中级语言系统(如 Python 或者 Perl)基本不会对你有太多帮助。
(2) 精通那种语言。研究它产生的代码，具体做法是：将其加载到调试器，或者编译成汇编语言源并研究产生的汇编源代码文件。
(3) 学习如何编写和连接汇编语言函数到你所选择的高级语言所编写的程序之中。
(4) 研究底层的窗口机制。对于 Linux 而言，这将是 X Window 技术，关于它已经有很多非常好的书籍。我最喜欢的一本书是 *The Joy of X*，作者是 Niall Mansfield (Addison-Wesley 出版公司，1994)。
(5) 仔细研究一个特定的桌面环境和部件集细节，如 GNOME、KDE、xfce 或者其他环境。(有很多种环境，而且其中一些被设计为轻量级，相对来说比较简单，如 Xfce 和 WindowLab。)做到这一点的最佳办法就是用你所选择的高级语言为它编写应用程序，然后再研究汇编语言编译器产生的代码。
(6) 最后，通过模仿编译器生成的内容，尝试构建你自己的汇编代码。

不要指望能够找到很多在线帮助。Unix(及 Linux)在可移植方面投资很多，这就要求其中大部分操作系统及为其编写的所有应用程序只需通过简单的重新编译就能移植到一个全新的硬件平台。汇编语言是在 Unix 世界里最不受欢迎的孤儿(不受欢迎的程度几乎与 Pascal 相当，虽然 Pascal 是我自己最喜欢的高级语言)，许多文化部落主义会劝你不要企图雄心勃勃地使用汇编语言做任何事情。要坚持自己的信念，但是请记住，你在很大程度上必须依靠自己。

如果只是在寻找一种对汇编语言的更先进的挑战，可以研究使用 Unix 套接字编写网络应用程序。这不需做大量的研究，利用该技术制作的应用程序可以用于管理服务器或者其他根本不需要图形用户界面的后台软件包。在套接字编程方面有几本好书，其中大部分是由 Richard Stevens 编写的。认真研读，这是一项令人着迷的事情。

6.3　使用 Linux Make

如果曾经用过 C 语言编写程序，那么你应该熟悉 Make 实用程序的思想。Make 机制成长于 C 世界，虽然它已经被许多其他的编程语言和环境所接纳，但

是从未像 C 世界里接纳得那样彻底。

Make 机制所做的事情就是由其组成部分生成可执行程序文件。Make 实用程序是一个木偶主人，它根据总体规划(它是一个简单的文本文件，我们将其称为“生成文件”)来执行其他程序。在指定如何完成某项任务方面，生成文件(默认情况下，名为“makefile”)有点像计算机程序，但它没有精确地指定即将采用的操作序列，这一点又不太像计算机程序。它所做的事情就是指定需要用程序的哪些部分来构建程序的其他部分，这样做最终定义了它采用什么来构建最后的可执行文件。它通过指定某些名为依赖条件(dependency)的规则来完成该项任务。

6.3.1 依赖条件

纵观本书，我们看到的都是一些极小的程序，只有 250 行甚至更少的代码。在现实世界中，凡是具有一些实用价值的程序都可能拥有成千上万行，甚至数百万行源代码。管理这样数量巨大的源代码是软件工程的核心问题。以模块化方式编写程序是最古老也是最常用的处理程序复杂性的方法。将一个较大的程序切割成较小的块，然后单独处理每一块，将为你带来很大的帮助。

在一些大型程序中，某些模块被进一步分割成更小的块，甚至有时各个模块采用不止一种编程语言编写。当然，这就带来另一项挑战：了解如何创建这些模块以及如何将它们组合在一起。为此，你真的需要一个计划。

生成文件就是这样的蓝图。

在一个模块化程序中，每一个代码块都以某种方式被创建，一般情况下通过使用编译器或者汇编编译器和连接器来实现。编译器、汇编编译器和连接器采用一个或多个文件，并由它们来创建新的文件。如前文所述，汇编编译器采用一个充满汇编语言源代码的.ASM 文件，并用它来创建一个可连接的目标代码文件。如果根本没有或者不处理源代码文件，将无法创建目标代码文件。目标代码文件的存在依赖于源代码文件。

同样，连接器把多个目标代码文件连接在一起，形成一个可执行文件。可执行文件的存在依赖于目标代码文件。生成文件(makefile)的内容指定了构建哪些文件需要哪些其他文件，以及需要采取哪些步骤来实现。Make 实用程序查看生成文件中的规则(依赖条件)并调用任何编译器、汇编编译器和它认为构建最终可执行文件或者库文件所需的其他实用程序。

有很多种生成(Make)工具风格，并非所有的生成文件都能被随处可见的其他生成工具所理解。然而，Unix 的 Make 实用工具是相当标准的，这里要讨论的就

是 Linux 附带的 Make 实用工具。

让我们举一个例子，该例子实际上生成一个简单的 Linux 汇编程序。通常，在创建一个生成文件时，首先要确定哪个文件或者哪些文件是创建可执行程序文件所必需的。可执行文件是在连接步骤被创建的，所以第一个必须定义的依赖条件是连接器需要哪些文件来创建可执行文件。依赖条件本身的声明可以十分简单：

```
eatsyscall: eatsyscall.o
```

这只是说，为了生成可执行文件 eatsyscall(第 5 章程序清单 5-1)，我们首先需要拥有文件 eatsyscall.o。上述的行实际上是一个依赖条件，生成文件中应该包含的就是这种形式。在任何程序中，除了最小的程序(比如本例)以外，连接器都将连接不止一个.O 文件。因此，一个可执行文件依赖于一个目标代码文件，这可能是最简单的依赖条件。如果必须连接其他文件以生成可执行文件，那么这些文件必须以列表形式列出，中间用空格隔开：

```
linkbase: linkbase.o linkparse.o linkfile.o
```

上面这行告诉我们，可执行文件 linkbase 依赖于三个目标代码文件，在生成想要的可执行文件之前，所有这些文件都必须存在。

像这样的命令行告诉我们哪些文件是必需的，但是并不告诉我们必须对它们做些什么。这是一个计划的重要组成部分，它由依赖条件后面的行负责处理。两个命令行一起工作。针对我们这个简单例子，有以下两行：

```
eatsyscall: eatsyscall.o
	ld -o eatsyscall.o eatsyscall
```

至少对于 Linux 版本的 Make 实用程序而言，第二行的行开头应该至少缩进一个制表符。我之所以强调这一点，是因为如果在第二行起始位置没有制表符，Make 实用程序将会报错。利用空格字符实现的缩进将不生效。一个典型的“missing Tab”的错误信息(初学者会看到很多这种提示)如下所示：

```
Makefile:2: *** missing separator. Stop.
```

这里，第二行的开始位置没有出现制表符。

生成文件的这两行代码合起来应该很容易理解：第一行告诉我们做这项任务需要哪个文件或者哪些文件，第二行告诉我们如何做这项任务。在本例中，通过

使用 Ld 连接器将 eatsyscall.o 连接到可执行文件 eatsyscall 中。

简单概括如下：我们指定哪些文件是必需的，以及必须对它们做哪些事情。但是，Make 机制还有一个非常重要的方面：知道实际上是否有必要执行某项任务。

6.3.2 文件何时最新

这么说可能有点白痴，但一旦一个文件已经被编译或连接，它就已经完成，并且不必再进行任何处理……直到你修改其中一个所需的源代码文件或者目标代码文件。Make 实用程序知道这一点。它知道编译或连接任务是否需要完成；如果不必做某项任务，Make 实用程序将拒绝执行它。

Make 实用程序如何知道是否需要做某项任务呢？考虑以下依赖条件：

```
eatsyscall: eatsyscall.o
```

Make 实用程序看到这一行，便明白可执行文件 eatsyscall 依赖于目标代码文件 eatsyscall.o，如果没有 eatsyscall.o，将无法生成 eatsyscall。它还知道两个文件的最后更改时间，因此，如果可执行文件 eatsyscall 比 eatsyscall.o 新，它便推断出对于 eatsyscall.o 所做的任何更改都已反映在 eatsyscall 中。(可以完全肯定，因为产生 eatsyscall 的唯一方式是通过处理 eatsyscall.o。)

Make 实用程序非常重视 Linux 的时间戳。无论什么时候编辑源代码文件，或者生成目标代码文件或可执行文件，Linux 都将该文件的时间戳更新为最终完成更改的时刻，即便是在几个月前就创建了原始文件，按照惯例，如果该文件的时间戳的值比另一个文件的时间戳的值更接近现在，就认为它比较新，即使另一个文件刚刚在 10 分钟前创建。(如果你对时间戳的概念比较陌生，可以这样来理解：它只是操作系统为每一个文件系统目录下的文件保留在系统目录中的值，每当一个文件被更改时，它的时间戳就被更新为当前的时钟时间。)

当一个文件比它所依赖的所有文件都新(根据该生成文件中调用的依赖条件)时，可认为该文件是最新的，重新生成它不会有任何收获，因为所有包含在成员文件中的信息已经被反映到依赖文件中。

6.3.3 依赖链

到目前为止，还没看出 Make 机制有太大的用途，但是当一个单独的生成文

件包含依赖链之后，Make 机制的真正价值才开始显现。即使在最简单的生成文件中，也会有某些依赖条件又依赖于其他依赖条件的情况。我们那个完全没有实用价值的示例程序需要在它的生成文件中声明两个依赖条件。

思考一下，下述依赖条件语句指定如何从一个目标代码(.O)文件产生一个可执行文件：

```
eatsyscall: eatsyscall.o
    ld -o eatsyscall.o eatsyscall
```

这里的要点是，生成 eatsyscall 文件要从 eatsyscall.o 开始，根据第二行的“配方”对它进行处理。那么，eatsyscall.o 从何而来？这就需要声明另外一个依赖条件：

```
eatsyscall.o: eatsyscall.asm
    nasm -f elf -g -F stabs eatsyscall.asm
```

在这里，我们将其解释为，要想生成 eatsyscall.o，我们需要 eatsyscall.asm，为了生成它，我们需要遵循第二行的“配方”。完整的生成文件将只包含这两个依赖条件：

```
eatsyscall: eatsyscall.o
    ld -o eatsyscall.o eatsyscall
eatsyscall.o: eatsyscall.asm
    nasm -f elf -g -F stabs eatsyscall.asm
```

这两个依赖条件语句定义了我们要想从非常简单的汇编语言源代码文件 eatlinux.asm 中生成一个可执行程序文件所必须采用的两个步骤。然而，仅从这里显示的两个依赖条件来看，这样做的价值还不是很明显。要想编译 eatlinux.asm，我们需要在某种程度上连接 eatlinux.o 来创建 eatlinux。这两个步骤几乎在所有情况下都是缺一不可的。

但是考虑一个包含数百个单独源代码文件的实际的编程项目。这些文件中只有一部分可能会在编辑器中“受苦”和在任意一天被修改。然而，为了创建和测试最后的程序，所有的文件都是必需的。这是否意味着所有的编译步骤和汇编步骤都是必需的呢？当然不是。

可执行程序是由连接器从一个或多个(更多时候是多个目标代码)文件编织而成的。如果(假设)除了两个文件之外，所有的目标代码文件都是最新的，那么就没有理由再编译一次其余的 147 个源代码文件。只需编译两个已更改的源代码文

件，然后再将所有的 149 个目标代码文件连接到可执行文件即可。

当然，挑战在于正确记住哪两个文件发生了变化，并且确保最近对所有 149 个源代码文件中的任意一个所做的任何修改都已经反映在最终的可执行文件中。这是很大的记忆量，或者可以参考一些笔记，而且当不止一人从事一个项目时，情况会越来越恶化，这是几乎所有商业软件开发企业的典型情况。Make 实用程序使我们不必记忆相关内容。它能够自己将其推算出来，并且只做那些必须做的工作，不多，也不少。

Make 实用工具查看生成文件及其调用的所有源代码文件和目标代码文件的时间戳。如果可执行文件比所有的目标代码文件新，那么什么事情也不需要做，但如果任何一个目标代码文件的版本比可执行文件新，那么该可执行文件必须重新连接。如果一个或多个源代码文件要么比可执行文件新，要么比它们各自的目标代码文件新，则在做任何连接工作之前，必须先对它们做一些编译或者汇编工作。

Make 实用工具所做的事情从可执行文件开始，然后开始寻找从该可执行文件中移走的依赖链。可执行文件依赖于一个或多个目标文件，而目标文件依赖于一个或多个源代码文件，Make 实用程序遍历各种链路构成的路径，留意什么比什么版本更新，以及要想把它处理好所必须做的事情。然后，Make 实用程序有选择地执行编译器、汇编编译器和连接器，以确保最终的可执行文件比它依赖的所有文件都新。务必确保需要做的所有工作都已经完成。

此外，Make 实用程序可以避免花费不必要的时间去编译和汇编那些已经是最新的因而没有必要再次编译或汇编的文件。由于在大型程序中，一个完整的生成过程(我指的是重新编译/重新汇编和重新连接每个项目中的单独文件)可能需要几个小时的时间，如果只需对程序的一小部分所做的更改进行测试，Make 实用程序将会节省大量的时间。

其实，Unix 的 Make 实用工具还有很多其他功能，这里描述的只是一些基本原则。你还可以制定一些编译条件、文件包含条件等。在刚刚开始学习汇编语言(或 C 语言编程)时，不必在这些事情上做文章。但是，随着编程技能的提高，以及参与的大型项目越来越多，知道有这种权利是非常有好处的。

6.3.4　从 Kate 编辑器内部调用 Make 实用工具

像你在编程过程中做过的所有其他事情一样，运行 Make 实用程序非常简单：在命令行中输入 make，然后按 Enter 键即可。Make 实用程序将处理其余的

事情。适合初学者兴趣的命令行选项只有一个，即-k。-k 选项指示 Make 实用程序停止生成任何发生错误的文件，并且留下未受干扰的目标文件的副本。(它继续生成其他需要生成的文件。)如果没有-k 选项，Make 实用程序可能会用不完整的副本覆盖现有的目标代码和可执行文件，这虽然不是世界末日，但是有时会带来一种滋扰和混淆。如果现在你还不是很明白，不用担心，那就一直使用-k，直到你确信真的不需要它。这就是说，对于那些每个目录下面只有一个项目并且生成文件名为“makefile”的简单项目而言，调用 Make 实用工具只需如下命令：

```
make -k
```

任何时候对任一源代码文件做了任何修改，无论多么微小，都必须运行 Make 实用工具来测试一下这种修改所带来的后果。作为初学者，你可能要通过“微调和尝试”的方法来学习，这将意味着你可能只修改源代码文件中的某一行代码的某一条指令，就得“看看那样做会带来什么”。

如果你的确倾向于以这种方式学习(并且这样做也并没有什么错！)，那么你将需要多次运行 Make 实用程序。著名的 Emacs 文本编辑器包含一个键绑定功能，能够让你只按一个按键即可运行 Make 实用程序。我们也可以在 Kate 编辑器中做同样的事情，这样，一将更改保存到一个文件，就可以马上通过按下某个按键来为你的项目打开 Make 实用程序了。

为了创建自己的 Make 实用程序快捷键，必须添加一个按键绑定，它不是绑定到 Kate 编辑器，而是绑定到 Konsole 终端仿真程序。Konsole 是嵌入 Kate 编辑器中的，所以当在源代码窗口中打开终端窗口时，实际上打开的是一个 Konsole 的副本。为 Konsole 定义的按键绑定会添加到 Konsole 的所有副本，包括嵌入 Kate 编辑器中的那个。

这些选项是深埋在 Konsole 菜单树里面的，所以请仔细阅读。

(1) 从桌面而不是从 Kate 编辑器内部启动 Konsole。

(2) 从 Konsole 主菜单选择 Settings→Manage profiles(管理配置文件)。

(3) 如果还没有配置文件，创建一个新的配置文件。在本章的前面部分，我讲述了如何为 Konsole 创建一个新的配置文件，用以提供 IBM-850 字符编码(用于老式的方块字符集)；如果你又想创建一个新的配置文件让编码方案变回去，选择并打开它即可。

(4) 当 Edit Profile(编辑配置文件)对话框出现时，单击 Input(输入)标记。

(5) 当 Key Bindings(按键绑定)对话框出现时，确保 xFree 4 被选中。在 Ubuntu 8.10 下使用 Konsole 时，这是默认设置。单击 Edit 按钮。

(6) 当 Edit Key Bindings List(编辑按键绑定列表)对话框出现时，向下滚动绑定列表直到在键组合列中看到 ScrollLock 键。我们将劫持 ScrollLock 键，我认为它是标准 PC 键盘上最没用处的按键。如果已经使用 ScrollLock 键做某件其他事情，就需要选择一个不同的键了。

(7) 双击 ScrollLock 键右侧的输出列。这样就能够输入一个字符串，在 Konsole 具有焦点的情况下，任何时候当 ScrollLock 键被按下时，该字符串都将被 Konsole 发送到标准输出。在输出列中，输入字符串：**make -k/r** (见图 6-11)。

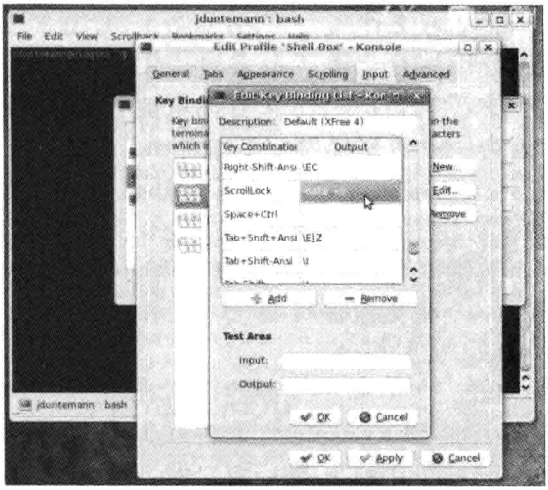


图 6-11　向 Konsole 添加一个按键绑定

(8) 在 Edit Key Bindings List(编辑按键绑定列表)对话框中单击 OK 按钮，并且在 Key Bindings(按键绑定)对话框中单击 OK 按钮。然后在 Manage Profiles(管理配置文件)对话框中单击 Close 按钮。大功告成！

可以打开 Konsole 并按下 ScrollLock 键测试一下这个新的绑定键。应该在 Konsole 命令行中键入 make -k，后面跟 Enter 键。(那是/r 在按键绑定字符串中的含义。) Make 实用程序将被调用，并且根据 Konsole 是否针对一个包含生成文件的项目目录打开，生成你的项目。

现在可以启动 Kate 编辑器，打开它的终端窗口，并且做同样的事情。按下 ScrollLock 键，终端窗口将调用 Make 实用程序。每次按下 ScrollLock 键，都将再次调用 Make 实用程序。

6.3.5 使用 touch 命令强制执行生成操作

如前所述，如果可执行文件比它所依赖的所有文件都新，Make 实用程序将拒绝执行生成操作——毕竟，在它对这个过程的理解中，当可执行文件比它所依赖的所有文件都新时，它就没有工作可做。

但是，偶尔也会有这样的情况：你想让 Make 实用程序执行一次生成过程，即使可执行文件是最新的。初学者最可能遇到的一种情况就是当“修理”生成文件本身时。如果修改了生成文件，并且想对它进行测试，但可执行文件本身又是最新的，那么，你就需要做一些有说服力的事情。Linux 有一个名为 touch 的命令，它只有一个任务：将一个文件的时间戳更新为当前时间。如果对源代码文件调用 touch 命令，该文件将会奇迹般地变为比可执行文件还要新，Make 实用程序将会乖乖地重新生成可执行文件。

在终端窗口中调用 touch 命令，后面跟需要处理的文件的名称：

```
touch eatsyscall.asm
```

然后，再次调用 Make 实用程序，将会执行生成过程——假设生成文件存在并且正确！

6.4 Insight 调试器

在第 5 章结束时，我曾带着你通过了一个简单的编辑/编译/连接/测试/调试开发周期。在该过程中，我使用的调试器是 KDbg，之所以选择它，是因为它不仅简单而且易用。但是 KDbg 却存在一个问题：它并不完全理解汇编语言可执行文件，而且不显示内存中的数据，这一点与对使用其他语言(例如 C 和 C++)编写的可执行文件所做的事情有所不同。所以，尽管 KDbg 在单指令调试过程中查看寄存器的值这方面是很有用的，但是它也不是一劳永逸的调试器，你很快就会跳起来反对它的汇编语言限制。(在写这本书时，KDbg 的在线帮助系统刚好发生了故障，所以无法访问帮助文件。)

有很多种调试器，也有很多像 KDbg 一样的调试器“前端”。KDbg 本身并不是一个真正的调试器。它是标准的 GNU 调试器——Gbd 的一个软件控制面板，而且是随着所有 Linux 版本自动安装的。当使用 KDbg 时，其实真正使用的是 Gdb，Gdb 是 Linux 软件开发环境的一个基础组件。Gdb 调试器没有用户界

面。它严格地在终端窗口使用纯文本工作，是我用过的最难使用的软件之一。它确实需要一些图形用户界面，这就是诸如 KDbg 和 DDD (数据显示调试器)之类的 Gdb 前端存在的原因。

Gdb 存在的时间几乎和 GNU 项目一样长，这可以追溯到 1985 年。大约 15 年前，有一些人认为未来是属于图形用户界面平台的，Gdb 调试器需要一个自己的图形用户界面。Insight 界面最初是由 Red Hat Linux 组织和 Cygnus Solutions(一个 GNU 支持公司)合作完成的。这两家公司于 1999 年合并，Red Hat 自那时起，开始继续进行 Insight 的研究工作。

Insight 与 KDbg、DDD 以及其他 Gdb 前端产品不同。Insight 实际上是 Gdb 的一部分，并且提供 Gdb 操作的另外一种(用于在图形化桌面如 GNOME、KDE、xfce 以及所有的其余的桌面系统上使用的)窗口视图。这种视图是综合的，如果同一时间在桌面上打开所有内容，可能有点吓人。然而好消息是，Insight 是模块化的，它能够在工作台上采用不同的窗口显示项目的不同视图。你可以关掉认为不需要的或者尚未理解的视图，这样，调试器看上去就不会再像原来那样令你手足无措了。

在本章的其余部分，我将会讲解 Insight 及其各种视图，以及如何将程序加载到其中并进行观察。当我在本书的其余部分提及“调试器”时，我所指的就是 Gdb/Insight 调试器。

6.4.1　运行 Insight

Insight 是随 Linux 安装的，并没有什么更多的内容需要寻找。可以从任意的终端仿真程序(如 Konsole)来运行它，只需在命令行中输入它的名字即可。在实践中，通常通过在命令行中 insight 的后面指定程序名，在运行 Insight 的同时加载被测试的程序。例如，要想使用程序清单 5-1 所示的程序，导航到 eatsyscall 项目目录，打开一个 Konsole 窗口，并通过以下方式运行 Insight：

```
insight eatsyscall
```

如果打算通过 I/O 重定向来调试程序，当调用 Insight 时，该重定向必须与可执行文件名一起输入命令行中：

```
insight eatsyscall > eattalk.txt
```

从什么终端窗口启动 Insight 并不重要。我的建议是在打开与打算调试的项目对应的 Kate 会话后，从 Kate 编辑器的终端窗口中打开它。当打开一个会话

时，Kate 编辑器导航到目录文件所在的会话，所以当启动它时，不必担心 Insight 能否找到可执行文件或者源文件。

6.4.2　Insight 的众多窗口

当第一次运行 Insight 时，可能会看到多达 10 个不同的窗口在屏幕上弹出来。(不同版本的 Insight 默认启动的窗口数量可能有所不同。)这 10 个窗口中的每一个都是正在调试的项目的不同视图。Insight 的主窗口称为源代码窗口。其余的 9 个窗口可以关掉，如果你想这样做的话；当你还是初学者时，只有其中 3 个窗口真正对你的汇编语言项目有用。

所有的 Insight 窗口都可以从 View 菜单中打开。每个窗口都可通过单击其右上角的 Close 按钮来单独关闭。表 6-2 总结了 9 个视图。为了跟上书中示例，只需源代码窗口、寄存器视图和存储器视图即可。

表 6-2　Insight 的视图

窗口	快捷键	用途
堆栈	Ctrl+S	汇总和导航已经存在的堆栈帧
寄存器	Ctrl+R	汇总 CPU 寄存器
内存	Ctrl+M	显示某一给定内存地址处的值
观察表达式	Ctrl+T	显示和编辑观察值
局部变量	Ctrl+L	显示和编辑堆栈帧中的局部变量
断点	Ctrl+B	汇总所有已经存在的断点
控制台	Ctrl+N	打开一个对 Gdb 命令行的接口
函数浏览器	Ctrl+F	查找函数并允许设置断点
线程列表	Ctrl+H	汇总属于可执行文件的线程

Insight 的窗口中有几个是 C 专用的，不一定会对你有很大帮助。查看和设置局部变量监视窗口取决于可执行程序中是否拥有汇编程序刚好不提供的调试信息。其他窗口不属于初学者的范围：线程列表视图只适用于多线程程序，这是一个我不能在本书中涵盖的高级话题。对于较短的程序，根本不需要断点视图，因为可以在源代码窗口中看到断点。控制台视图使你可以通过传统命令行界面来控制 Gbd，这种方式尽管很难掌握，但是可能以后会有用。

一旦开始定义可调用的过程时，堆栈视图会自己生效，特别是当我们开始从

标准的 C 库文件中调用函数时，我将在本书的结尾部分对这一点进行讲解。在接下来的几章中，将主要查看源代码、CPU 寄存器和内存。

如果拥有一个小巧的显示器，那么，在同一屏幕上排列 Kate 编辑器和 Insight 以便让这两者均可被完全看到，这也许需要一些技巧。(不要尝试在任何对角线小于 22 英寸的显示器上显示它。) 但是，这并不是一个问题，因为你要么修改程序，要么调试它，不能在同一时间内做这两件事情。所以在完成编辑工作并启动 Make 生成项目之后，请从 Kate 编辑器的终端窗口启动 Insight。一旦发现自己的项目中已经存在足够多的错误行为，关闭 Insight，并返回 Kate 编辑器，重新对其进行编辑。 Insight 的窗口状态是持久化的，下一次你运行 Insight 时，只有上次打开的窗口才会出现，并且会出现在显示器上的同一个位置。

唯一应该为 Insight 修改的重要配置选项是选择“Windows 风格的”工具栏图标集。选择 Preferences→Global，在出现的 Global Preferences(全局参数设置)对话框中，下拉 Icons(图标)菜单，选择 Windows 风格图标集。单击 OK 按钮。Windows 风格的图标更详细些，当你还在学习之路上探索时，它能够帮助你记起这些按钮所代表的含义。

6.4.3　快速体验 Insight

在第 5 章中，我解释了调试器如何单步执行程序，Insight 以同样的方式工作。让我们把程序清单 5-1 加载到 Insight 中，看看它能够做哪些事情，特别是观察内存数据这一方面。打开 Kate 编辑器以及 eatsyscall 项目的会话。确保该项目已经生成，并且在项目目录下存在一个可执行文件。然后启动 Insight。

```
insight eatsyscall
```

这一次，只需要主窗口视图(源代码视图)，再加上寄存器视图和存储器视图。关闭任何其他可能已经打开的视图。你所看到内容应该如图 6-12 所示。

Insight 窗口的外观不如 KDbg 窗口，但它的优点在于所有控件都有悬停帮助提示。当鼠标指针悬停在其中一个控件上时，将弹出一个标签，告诉你这是什么控件。Insight 的默认设置都不错，在单步调试一个程序之前通常不需做任何设置。

当加载一个可执行文件用于调试时，Insight 高亮显示第一条可执行指令。这看起来像是在断点处暂停执行，但是事实上并非如此：默认情况下没有设置任何断点，而且程序并不运行。其中的高亮显示只是告诉你，当运行该程序时，执行将从何处开始。

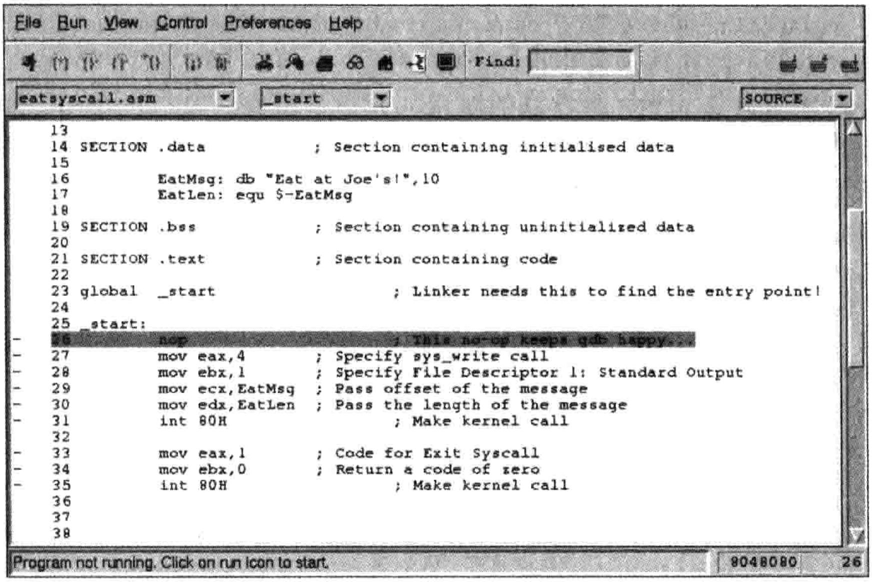


图 6-12　Insight 的源代码窗口

像在 KDbg 中一样，我们必须设置初始断点。可以通过单击源代码的最左侧列显示的虚线字符来设置一个断点。每一个包含指令的代码行中都有一条虚线，因此，可以接受一个断点。(不可以在一个数据定义上或者除了指令之外的其他内容上设置断点。)断点切换：单击一下虚线，将出现一个红色的正方形，即表明断点存在；单击红色的正方形，断点即被删除，又返回到虚线显示。

单击指令 `MOV EAX,4` 旁边的虚线，会得到一个断点。现在可以运行该程序了。Insight 的导航控件是工具栏最左边的两组按钮。可以通过单击“运行”图标(它显示为一名正在奔跑的男子)或者按下 R 键来启动该程序。Insight 将开始执行程序，然后暂停在断点处。此外，还将发生另外两件事情：

- Insight 将会把看似断点的东西放到程序的第一条指令上(在这里，是一条 NOP 指令)，但是它不会停在这里。高亮显示将会移动到初始断点上。其原因涉及 C 程序的内部机制，我不便在这里进行详细解释。
- 原本为空的寄存器视图和存储器视图一下子活跃起来，并显示真实数据。

像 KDbg 一样，Insight 真的是为 C 语言编程而设计的，与汇编语言相比，一些控件更多地应用于 C 语言。如果想单步调试简单的程序，请单击 Step Asm Instruction 按钮。高亮显示的指令将会得到执行，并且高亮显示依次移到下一条指令上。当开始编写含有子程序调用的代码时(我将在后面的章节中进行演示)，

用于指令单步调试的两个按钮是有所不同的，但是对于诸如 eatsyscall 之类的简单项目而言，两个按钮所做的事情完全相同。

寄存器视图显示 CPU 寄存器的状态，并且将所有在最后一条指令执行过程中发生改变的寄存器用绿色高亮显示。继续单步运行该程序，直到高亮显示出现在第一条 `INT 80H` 指令上。4 个寄存器已经加载了数据，并为系统调用做好准备，可以通过寄存器视图查看它们，如图 6-13 所示。

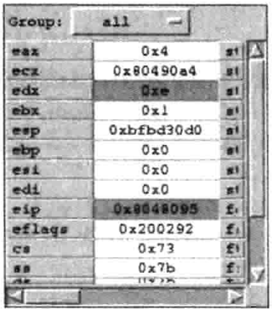


图 6-13 程序执行过程中的 Insight 寄存器视图

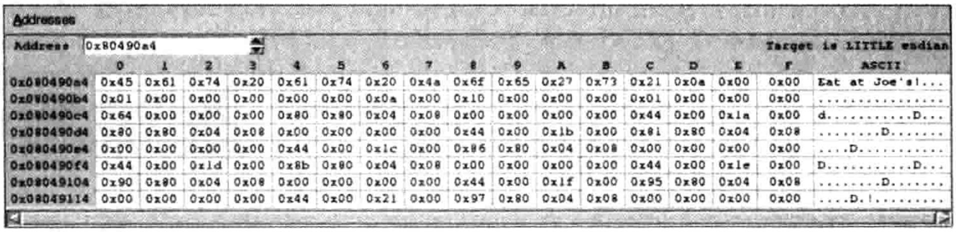
EDX 寄存器被用绿色高亮显示，因为它是在 Insight 执行的最后一条指令中加载的。EIP 寄存器(指令指针寄存器)在几乎所有的时间内都是绿色，因为只要有一条指令被执行，它就会发生改变，也就是说，在每一步中。

图 6-13 所示的寄存器视图窗口已经被大大缩小了。CPU 拥有很多寄存器，寄存器视图将乖乖地显示所有寄存器的内容。可以通过最大化寄存器视图窗口来查看整个寄存器列表。随着窗口的最大化，Insight 将会显示段寄存器(段寄存器属于操作系统，你不能使用)，以及在学习汇编的第一阶段不会用到的算数处理器寄存器。也可以选择只显示通用寄存器，这是通过单击寄存器视图顶部的 Group 按钮，然后从出现的菜单中选择 General 来实现的。还可以通过改变窗口的大小来实现同样的效果，具体采用哪种方式随你选择。

当程序开始运行时，存储器视图窗口加载从.data 段的第一个已命名数据项开始的数据。在程序清单 5-1 中，只有一个数据项，即 EatMsg 字符串。在显示数据的起始位置，将会看到“Eat at Joe’s”这一消息(见图 6-14)。

像寄存器视图一样，存储器视图在每条指令执行后都会更新。在那些在内存中操作数据的功能更强大的程序中，我们可以看到，内存的内容随着逐步通过程序的机制而发生变化。Insight 的存储器视图很像第 5 章中介绍过的 Bless 十六进制编辑器。内存以两种形式显示：在左边是一些由 16 个独立的十六进制数值构

成的行，而在右边，是一些由 16 个 ASCII 字符组成的行。那些不可显示的 ASCII 字符数据值被显示为句号字符。



Addresses

Address 0x80490a4　　Target is LITTLE endian

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
0x080490a4	0x45	0x61	0x74	0x20	0x61	0x74	0x20	0x4a	0x6f	0x65	0x27	0x73	0x21	0x0a	0x00	0x00	Eat at Joe's!...
0x080490b4	0x01	0x00	0x00	0x00	0x00	0x00	0x0a	0x00	0x10	0x00	0x00	0x00	0x01	0x00	0x00	0x00	....
0x080490c4	0x64	0x00	0x00	0x00	0x80	0x80	0x04	0x08	0x00	0x00	0x00	0x00	0x44	0x00	0x1a	0x00	d...........D...
0x080490d4	0x80	0x80	0x04	0x08	0x00	0x00	0x00	0x00	0x44	0x00	0x1b	0x00	0x81	0x80	0x04	0x08	........D.......
0x080490e4	0x00	0x00	0x00	0x00	0x44	0x00	0x1c	0x00	0x86	0x80	0x04	0x08	0x00	0x00	0x00	0x00	....D...........
0x080490f4	0x44	0x00	0x1d	0x00	0x8b	0x80	0x04	0x08	0x00	0x00	0x00	0x00	0x44	0x00	0x1e	0x00	D...........D...
0x08049104	0x90	0x80	0x04	0x08	0x00	0x00	0x00	0x00	0x44	0x00	0x1f	0x00	0x95	0x80	0x04	0x08	........D.......
0x08049114	0x00	0x00	0x00	0x00	0x44	0x00	0x21	0x00	0x97	0x80	0x04	0x08	0x00	0x00	0x00	0x00	....D.!.........

图 6-14　程序执行过程中的 Insight 的存储器视图

存储器视图还有另外一个巧妙的用处：可以在任意时间更改内存中的数据。可以在视图中的十六进制值部分进行修改，也可以在视图中的 ASCII 部分进行修改。试一试：在 ASCII 视图中单击，使光标刚好出现在“Joe's.”中的“e”的后面。退格三个字符，键入“Sam”，然后按 Enter 键。你已经在内存中将“Joe”改为了“Sam”。(磁盘上的文件不会受影响。)

要想验证那些更改，单击 Continue(继续)按钮(刚好与单步调试按钮形成对照)，继续不停地执行，一直到下一个断点，或者直至该程序结尾，具体看哪一个先到。(Continue 按钮是主工具栏上从左边数第 5 个按钮。)程序将运行到结束，并且如果在 Kate 编辑器的终端窗口中观察程序的输出，你将会看到自己已经劫持了 Joe 的广告口号，并将其赋给了 Sam。

6.4.4　拿起你的工具

此时此刻，你已经拥有了所需的背景知识和所需的工具。是时间坐下来(终于等到这一刻了！)，开始详细研究 x86 指令集，然后开始认真地编写程序了。

第 7 章

跟踪指令

与机器指令亲密接触

正如喜剧演员比尔·考斯比曾经说过的那样：我给你讲过那个故事，所以我也可以给你讲这个故事……至此本书的讲解已经完成超过三分之一，但是我还没有开始详细介绍 PC 汇编语言的主要元素：x86 指令集。大多数汇编语言书籍，即使是入门级别的，都愿意从指令集开始讲解，根本不考虑大多数人的基础，而如果没有这些基础，那些刚刚开始学习的程序员们将会完全迷失，最终选择放弃。

定位是至关重要的。这就是我在真正开始之前，采用 200 页的篇幅才到达其他书籍开始的地方的原因。

请记住，本书的创作目的是为你第一次接触汇编语言提供一些必要的基础，并指明方向。它不是 PC 汇编语言的完整课程。一旦“跑到”了本书的最后，你就可以抬脚奔向各个出版社的众多其他汇编语言书籍了。

进入本质的时刻到了，让我们一路向下，到达软件与硅片相遇的地方。

7.1 为自己建立一个沙盒

认识 x86 机器指令的最佳方法是建立一个自己的沙盒，并且玩得开心。沙盒中的汇编语言程序并不需要在 Linux 下正确地运行。它甚至不需要像一般程序那样完整。所有它必须具有的特点就是能够被 NASM 和连接器理解，而这本身并不需要做很多的事情。

用我个人的技术术语来说，沙盒是一个打算只在调试器中运行的程序。如果想看看一条指令对内存或者寄存器有什么影响，在 Insight 中对其进行单步调试即可充分了解。沙盒程序不需在命令行返回任何可视的结果，只要包含格式正确

的指令就可以。

在实践中，我的沙盒思想是这样的：为一个名为 sandbox.asm 的程序创建一个用于编译和连接的生成文件。你可以创建一个代码量很小的 NASM 程序，并将其以文件名 newsandbox.asm 保存到磁盘上。什么时候想“摆弄”机器指令，你都可以打开 newsandbox.asm，将其再次另存为 sandbox.asm，覆盖所有可能存在的 sandbox.asm 早期版本。你可以添加指令用于观察，并使用 Linux 的 Make 实用工具生成一个可执行文件。然后，将该可执行文件加载到 Insight 中，一次执行一条指令，在各种各样的 Insight 视图中仔细观察每一条指令都做了什么。

你的实验有可能产生一些具有保存价值的有用的机器指令组合。在这种情况下，将该沙盒文件另存为 experiment1.asm(或者任何你想赋予它的描述性名称)，一旦时机成熟，随时都能从那个序列生成一个“真正的”程序。

7.1.1 一个最小的 NASM 程序

那么，一个能够被 NASM 编译的程序需要具有哪些组成部分呢？事实上，内容并不多。清单 7-1 是我用于起步阶段沙盒的源代码。它所提供的内容实际上比 NASM 技术要求的还要多一些，但也只能算是一个比较有用的沙盒。

清单 7-1：newsandbox.asm

```
section .data
section .text

    global _start

_start:
        nop
; 将你的实验内容放在两个 nops 指令之间进行……

; 将你的实验内容放在两个 nops 指令之间进行……
        nop

section .bss
```

实际上，NASM 将编译一个根本不包含任何助记符指令的源代码文件，但平心而论，不包含任何指令的可执行文件将不会在 Linux 上运行。我们真正需要的是一个标记为全局(global)的起始点，这里为_start 标号。我们还需要定义一个如上所示的.data 段和一个.text 段。.data 段用于保存已命名数据项，当程序运行

时，它们将被赋予初始值。清单 5-1 中的“Eat at Joe’s”广告信息就是数据段中的一个已命名数据项。.text 段用于存放程序代码。要想创建一个可执行文件，这两个段都是必不可少的。

第二个标记为.bss 的段并不是绝对必要的，但是，如果打算进行实验，拥有它会带来一些好处。.bss 段保存未被初始化的数据，也就是说，该空间用于存放当程序开始运行时并没有接受任何初始值的数据项。从根本上来讲，这些都是空的缓冲区，用于存放程序运行期间即将产生的数据，或者从某处读取的数据。按照惯例，.bss 段位于.text 段的后面(我将在后面的章节中谈论更多有关.bss 段和未初始化数据的相关内容)。

要想使用 newsandbox.asm，先在 Kate 编辑器中创建一个名为 sandbox 的会话，并将 newsandbox.asm 文件加载到该会话中。然后立即将其另存为 sandbox.asm，这样就不必修改 newsandbox.asm 了。创建一个如下所示的生成文件：

```
sandbox: sandbox.o
    ld -o sandbox sandbox.o
sandbox.o: sandbox.asm
    nasm -f elf -g -F stabs sandbox.asm -l sandbox.lst
```

(解压缩本书的清单存档时，将会创建 sandbox 目录，该目录下已经有了这个文件和程序清单 7-1。)

在 sandbox.asm 中有两个 NOP 指令，它们出现在那里的目的是为了让我们在调试器中观察程序变得更加容易。要想研究机器指令，将它们放到两个注释之间就可以了。使用 Make 实用工具生成可执行文件，并将该可执行文件加载到 Insight 中：

```
insight sandbox
```

在两个注释之间的第一条指令上设置断点，然后单击 Run 按钮。执行开始，并在断点处停止。为了观察该指令的影响，单击 Step Asm 按钮。这也是第二个 NOP 指令位于那里的原因：当单步执行一条指令时，必须在该指令后面存在一条用于暂停执行的指令。如果沙盒中的第一条指令就是最后一条指令，那么在第一个单步执行中，程序执行就会“运行到边缘”，然后终止。当这种情况发生时，Insight 的寄存器视图和存储器视图将一片空白，根本看不到该指令的任何影响！

“运行到程序的边缘”是一个非常有趣的概念。如果单击 Continue 按钮，会

看到该程序没有正确结束时将会发生的事情：Linux 将提交一个分段错误，其中的原因可能有若干种。但是，本例中发生这种情况的原因是程序试图执行一个已经超过.text 段结尾位置的代码。当程序被加载时，Linux 知道该程序有多长，它不允许执行任何在程序中不存在的指令。

当然,这并没有什么持久性的伤害，Linux 非常善于处理错误的行为和畸形的程序，没有什么意外事件会对 Linux 有所影响。可以通过从 Insight 主菜单中选择 Run→Kill 来避免产生分段错误。Kill 命令只做一件事，那就是停止正在调试的程序，哪怕该程序暂停于断点，或者正在进行单步调试。

7.1.2 指令及其操作数

在汇编语言工作中，最常见的活动就是将数据从这里移到那里。有几种专门做这件事情的方法，但是真正通用的只有一个：MOV 指令。MOV 可以从一个寄存器到另一个寄存器、从一个寄存器到一个内存地址或者从一个内存地址到一个寄存器移动一个字节(8 位)、一个字(16 位)或者一个双字(32 位)的数据。MOV 所不能做的事情是将数据直接从一个内存地址移到另一个内存地址中。(要做到这一点，需要两条单独的 MOV 指令，一条用于将数据从内存中存入寄存器，另一条用于将数据从寄存器中存回到内存。)

MOV 这个名字有点名不副实，因为实际情况是：数据被从源复制到目的地，但是，一旦复制到目的地后，数据并不会从源上消失，而是继续存在于两个地方。这一点与我们直觉中的移动有点冲突，因为移动通常意味着某些东西从源上消失了，然后在目的地重新出现。

7.1.3 源操作数和目标操作数

大多数机器指令(包括 MOV)有一个或多个操作数 (有些指令没有操作数，或隐式地操作寄存器或存储器。当遇到这种情况时，我会在正文中特别强调)。思考如下机器指令：

```
mov eax,1
```

上述指令中有两个操作数：第一个是 EAX，第二个是数字 1。在汇编语言中，一条机器指令的第一个(最左边)操作数是目标操作数。从左边数第二个操作数是源操作数。

对于 MOV 指令而言，两个操作数的感觉相当忠于原义：将源操作数复制到目标操作数。在前面的指令中，源操作数(字面常量 1)被复制到目标操作数 EAX 中。在某些其他指令中，源和目的地的感觉并非如此，而是采用这样一种主要方式：只要机器指令导致产生一个新值，这个新值就被放在目标操作数中。

有三种不同类型的数据可以作为操作数：内存中的数据、寄存器中的数据和立即数。我已在表 7-1 中列举并解释了 MOV 指令的一些示例，你可以体验一下如何为 MOV 指令指定各种不同类型的操作数。

表 7-1 MOV 及其操作数

机器指令	目标操作数	源操作数	操作数注释
MOV	EAX,	42h	源操作数是立即数
MOV	EBX,	EDI	两个操作数都是 32 位寄存器数据
MOV	BX,	CX	两个操作数都是 16 位寄存器数据
MOV	DL,	BH	两个操作数都是 8 位寄存器数据
MOV	[EBP],	EDI	目标操作数是 32 位内存数据，该内存的地址存放在 ebp 中
MOV	EDX,	[ESI]	源操作数是 32 位内存数据，该内存的地址存放在 ESI 中

7.1.4 立即数

表 7-1 中的 MOV EAX,42h 是一个很好的立即数使用示例，它通过名为立即数寻址的寻址模式来访问立即数。立即数寻址的名字源于这样的事实：被寻址的项刚好是内置在机器指令中的数据。它不是存放在寄存器中，也不是存放在位于指令之外的某个内存中。立即数总是位于正被取出和执行的指令的内部。

立即数的尺寸必须与操作数相匹配。例如，不能将一个 16 位立即数存入一个 8 位的寄存器中(例如 AH 或者 DL)。NASM 不允许汇编如下的指令：

```
mov cl,067Efh067EFh
```

CL 是一个 8 位寄存器，而 067EFh 是一个 16 位的量。这样做不可以！

因为立即数刚好内置在机器指令内部，你可能认为它的访问速度会很快。这只在一点上是正确的：从内存中获取任何数据的时间均超过从寄存器中获取任何数据的时间，毕竟，指令都存储在内存中。因此，对立即数寻址的速度总是比对

存储在内存中的普通数据寻址的速度快，但是，从这两个地方取出数据的速度，都不如从 CPU 寄存器中简单地取出一个数值的速度快。

同时要记住，只有源操作数可以是立即数。目标操作数是数据要去的地方，而不是数据来自的地方。由于立即数由字面常量(数字，例如 1，0，42 或 07F2Bh)组成，任何试图将数据复制到立即数中而不是从立即数中复制数据的操作都毫无意义，并且始终是错误的。

NASM 中允许一些有趣的立即数形式。例如，以下的指令是完全合法的，只是它不像乍看起来那么有用：

```
mov eax,'WXYZ'
```

这是一条很好的指令，将其加载到你的沙盒里并在调试器中执行。在寄存器视图中查看寄存器 EAX 中的内容：

```
0x5a595857
```

这似乎有些不可思议，但是仔细看看：这个数字值与大写的 ASCII 字符 W，X，Y，Z 等值，并且已经被完全加载到 EAX 中。如果对 ASCII 码不是很熟悉，可查看一下附录 B 中的图表，W 是 57h，X 是 58h，Y 是 59h，而 Z 是 5Ah。每个字符的大小均为 8 位，因此其中 4 个字符刚好适合存放到 32 位寄存器 EAX 中。但是，它们是从后往前存放的！

嗯，不是这样的。回想一下早在第 5 章中介绍过的“字节序”的概念(如果不记得了，回去再读一下那一节)。x86 体系结构是“小端”字节序，这意味着在一个多字节序列中，最低有效字节存储在最低的地址。这一点同样也适用于寄存器，并且一旦知道如何引用寄存器内部的存储单元后，了解这一点会更有意义。

混乱是由我们从左至右的文字阅读习惯造成的。仔细看一下图 7-1。将其看作一个文本字符序列，“WXYZ”中的“W”被认为是最低有效元素。然而，EAX 是一个存放数字的容器，其中的最低有效列 (对西方语言而言) 永远在最右边。我们将 EAX 中的最低有效字节称为 AL，那是 W 的位置，将第二个较低的有效字节称为 AH，那是 X 的位置。EAX 中的两个最高有效字节没有单独的名称，不能对其进行单独寻址，但是它们仍然是 8 位字节，能够包含如 ASCII 字符一样的数值。序列“WXYZ”中的最高有效字符是“Z”，它被存放在 EAX 的最高有效字节中。

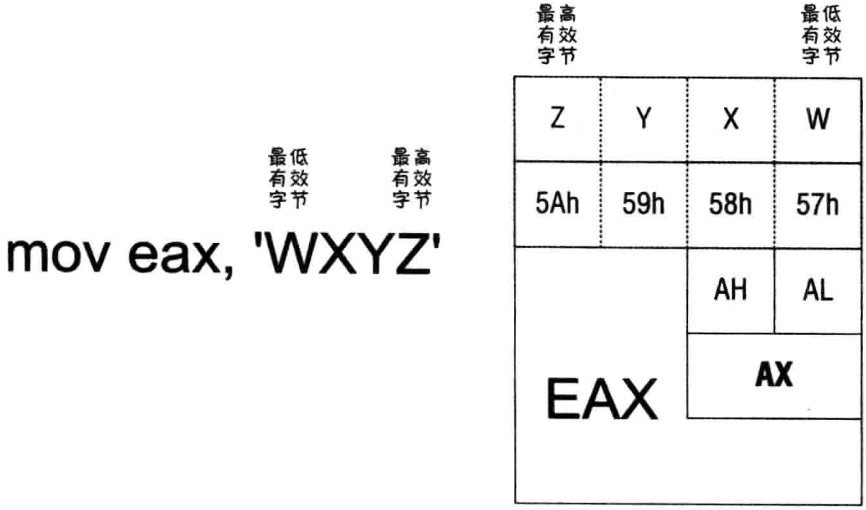


图 7-1　当作立即数处理的字符串

7.1.5　寄存器数据

我们将存储在 CPU 寄存器内部的数据称为寄存器数据，将直接访问寄存器数据的内存寻址模式称为寄存器寻址。寄存器寻址通过简单地指出想要工作的寄存器名来完成。下面是一些完全合法的寄存器数据和寄存器寻址示例：

```
mov ebp,esi          ; 32 位
mov bl,ch            ; 8 位
add di,ax            ; 16 位
add ecx,edx          ; 32 位
```

最后两个例子特别强调了一个事实：这里谈论的不只是 MOV 指令。正如预期的那样，ADD 指令所做的事情就是将源操作数和目标操作数相加，然后用求得的和替换目标操作数中原来的值。无论是什么指令，只要直接作用于寄存器中的数据，就会发生寄存器寻址。

汇编编译器跟踪某些没有什么实际意义的特定内容，其中一种情况就是在同一条指令中出现一个 16 位的半寄存器和一个完整的 32 位寄存器。这种操作是不合法的——毕竟，将一个 4 字节的源操作数移动到一个 2 字节的目标操作数到底意味着什么呢？将一个 2 字节源操作数移动到一个 4 字节的目标操作数中似乎是可能的，甚至是合理的，但是 CPU 并不支持这种操作，因此不能直接执行这种

操作。

在调试器中观察寄存器中的数据是深入了解其工作原理的最佳方式，尤其是刚刚开始学习汇编语言时。让我们来做点练习。将以下指令输入你的沙盒中，然后生成可执行文件，并将该沙盒的可执行文件加载到调试器中：

```
mov ax,067FEh
mov bx,ax
mov cl,bh
mov ch,bl
```

在 4 条指令中的第一条指令处设置第一个断点，然后单击 Run 按钮。单步执行这 4 条指令，仔细观察 AX、BX 和 CX 寄存器部分会发生什么。(记住，Insight 的寄存器视图并不单独显示 8 位或 16 位寄存器部分。AX 是 EAX 寄存器的一部分，CL 是 ECX 寄存器的一部分，等等。) 完成后，选择 Run→Kill 来终止程序。记住，如果选择 Continue 或者尝试超过程序的最后一步时，Linux 会提交一个分段错误，因为程序没有正确终止。任何东西在这样的故障中都不会受到伤害；记住，这个沙盒本就不是一个完整且正确的 Linux 程序。但是，通过 kill 终止程序而不是产生一个故障，这也是一种很好的实践体验。

下面是上述指令的功能总结：第一条指令是一个使用 16 位寄存器实现立即数寻址的例子。16 位十六进制值 067FEH 被移到寄存器 AX 中。第二条指令使用寄存器寻址将寄存器数据从 AX 复制到 BX 中。

第三条和第四条指令都在 8 位的寄存器段之间而不是在 16 位的寄存器段之间移动数据。这两条指令完成某些有趣的事情。看看最后一个寄存器显示，比较一下 BX 和 CX 中的值。通过将数据从 BX 移到 CX 中，每次移动一个字节，就有可能颠倒组成 BX 的两个字节的顺序。BX 的高半部分(有时也称为 BX 的最高有效字节，或 MSB)移进了 CX 的低半部分。而 BX 的低半部分(有时称为 BX 的最低有效字节，或 LSB)则进入了 CX 的高半部分。这是一个可以用在各种通用寄存器中的简单技巧。

为了防止你误认为 MOV 指令应该用于交换两个 16 位寄存器的一半，我建议你做下述内容。回到 Kate 编辑器，并在沙盒程序的结尾处添加下面的指令：

```
xchg cl,ch
```

重新生成沙盒，然后返回调试器，看看会发生什么。XCHG 指令将交换它的两个操作数的值。之前互相交换的值再次被互换，CX 中的值将和已经存在于 AX

和 BX 中的值相匹配。建议在首次编写汇编语言程序时：定期仔细检查指令集，以理解用四五条指令拼凑起来的程序是不可能使用一条指令实现的。x86 指令集非常擅长在这方面愚弄你。

在此提醒大家注意：有时候，在机器执行时间方面，某种“特殊情况”要比一般情况更快一些。例如被 2 的幂数相除，我们可以使用 DIV 指令实现，但也可以通过 SHR(右移位)指令实现。DIV 是更为一般的办法(你可以用它来实现任何无符号整数除法，不仅仅是 2 的幂数)，但是它的速度非常缓慢，在某些种类的 x86 处理器上，甚至比 SHR 指令的速度慢高达 10 倍！(本章后面部分将详细讲解 DIV 指令。)

7.1.6　内存数据

立即数是内置在机器指令本身内部的数据。寄存器数据是存放在某个位于 CPU 内部的寄存器中的数据，而内存数据是存储在分配给某个程序的系统内存中的某个 32 位内存地址处的数据。

除了一两个重要例外(如字符串指令，本章后文会涉及，但并不进行详细讲解)之外，一条指令的两个操作数中只有一个可以指定内存位置。换句话说，可以将立即数移到内存中，或将一个内存数据移到寄存器中，或其他的类似组合，但是不能直接将一个内存值移到另一个内存值中。这是当代 x86 CPU 所固有的限制，我们不得不接受它，尽管它有时会带来不便。

如果想要指定的数据是位于寄存器中的内存地址中的值，而不是寄存器本身中的数据，需要在寄存器名字的两边加上方括号。换句话说，如果想将存放在寄存器 EBX 中的内存地址处的字移到寄存器 EAX 中，应该使用以下指令：

```
mov eax,[ebx]
```

方括号中可以包含不止一个 32 位寄存器的名字，我们将在后面部分对其进行详细介绍。例如，可以在寄存器的括号内添加字面常量， NASM 会进行数学计算：

```
mov eax,[ebx+16]
```

也可以添加两个通用寄存器，就像这样：

```
mov eax,[ebx+ecx]
```

而且，如果这还不够，可以添加两个寄存器，外加一个字面常量：

```
mov eax,[ebx+ecx+11]
```

当然，这还没有包含全部的合法形式。括号中的所有内容称为内存数据项的有效地址(effective address)，并且有一些规则来规定哪些是合法的有效地址，哪些不是。在目前的 x86 硬件演变中，可以将两个寄存器加在一起，形成有效地址，但没有三个寄存器或更多的情况。换句话说，下面的指令是非法的有效地址形式：

```
mov eax,[ebx+ecx+edx]
mov eax,[ebx+ecx+esi+edi]
```

对于形式更为复杂的有效地址而言，演示要比解释更容易些，但是我们必须先了解一些其他内容。当处理查找表时，它们尤其有用，我将在后面部分对其进行更加深入的讲解。至于现在，最重要的事情是*不要*将一个数据项和它所存在的位置搞混淆！

7.1.7　混淆数据和它的地址

这听起来很乏味，但是请相信我，人们非常容易做这件事。回到程序清单 5-1，我们有如下的数据定义和指令：

```
EatMsg: db "Eat at Joe's!"
 . . .
mov ecx,EatMsg
```

如果曾经使用过任何一种高级语言，如 Pascal，你的第一直觉可能会认为存储在 EatMsg 中的所有数据会被复制到 ECX 中。但在汇编语言中并不是这样。MOV 指令实际上复制的是 EatMsg 的地址，而不是存储在 EatMsg 中(实际上，存储在 EatMsg 处)的内容。在汇编语言中，变量名代表的是地址，不是数据！

那么，实际上该如何到达一个由(像 EatMsg 一样的)变量表示的数据呢？同样，这也是通过使用方括号来实现的：

```
mov edx,[EatMsg]
```

这条指令所做的事情就是从 EatMsg 代表的内存地址中取出前 32 位数据，并从最低有效字节开始，将其加载到 EDX 中。根据我们为 EatMsg 定义的内容，那

将是“E”、“a”、“t”和“ ”4 个字符。

7.1.8　内存数据的尺寸

如果只想要其中一个单独字符，而不是前 4 个字符，该怎么办呢？如果不想要所有这 32 位呢？从根本上来讲，如果想使用一个字节的数据，需要将其加载到一个字节大小的容器中。寄存器 EAX 的大小是 32 位。但是，可以通过 AL 来寻址 EAX 的最低有效字节。AL 的大小是一个字节，我们可以通过使用 AL，采用以下方式带回 EatMsg 的第一个字节：

```
mov al,[EatMsg]
```

当然，AL 是包含在 EAX 中的——它不是一个单独的寄存器(如果这对你来说还不是很清楚的话，请参照图 7-1)。但是，通过 AL 这个名字，可以一次只从内存中取出一个字节。也可以使用 AX 这个名字来指代 EAX 中的低两字节(16 位)来实现类似的“特技”：

```
mov ax,[EatMsg]
```

这一次，字符“E”和“a”被从内存中读取出来，放到 EAX 的两个最低有效字节中。

将寄存器中的数据写出到内存时，尺寸问题变得有些棘手。NASM 并不像高级语言那样“记得”变量的大小。它只知道 EatMsg 从内存中什么地方开始，就这么简单。你必须告诉 NASM 需要移动多少个字节的数据。这是通过大小说明符(size specifier)来实现的。例如：

```
mov [EatMsg],byte 'G '
```

这里通过大小说明符 byte 告诉 NASM，你只想移出一个字节到内存中。其他的大小说明符包括 WORD (16 位)和 DWORD (32 位)。

7.1.9　糟糕的过去

很高兴你是在现在学习 x86 汇编，因为在前些年学习它要复杂得多。在 DOS 实模式下，对于一个有效地址的组成部分有几个限制，而这些限制在当今的 32 位保护模式下已经不复存在了。在实模式下，只有某些 x86 通用寄存器能够存放

内存地址：BX、BP、SI 和 DI。而其他的寄存器 AX、CX 和 DX 则不能。

更糟糕的是，正如第 4 章中描述的那样，一个地址有两个组成部分。我们必须考虑地址位于哪个地址段，当段地址不明确时，必须确保通过使用类似[DS:BX]的结构指定段地址。而且必须处理名为 ASSUME 的恶魔般的事情，对此还是少说为妙。(如果出于某些原因，你被迫在实模式段模型下为 x86 编写程序，请尝试找到本书的 2000 年版本，我在书中详细讲解了全部内容。)

在许多方面，我们的生活都变得越来越美好了。

7.2 CPU 的标志位

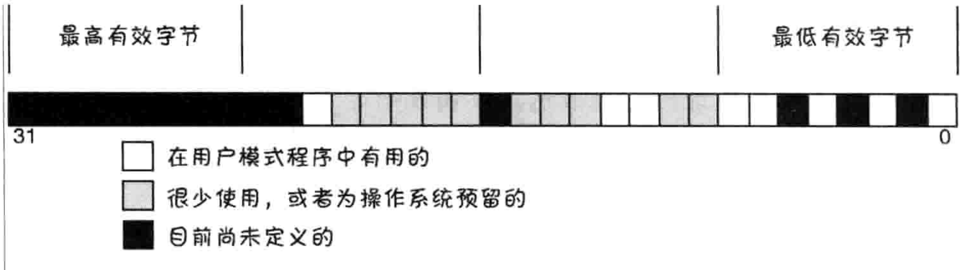
虽然我在 x86 体系结构概述中提到过 EFlags 寄存器，但是没有对它进行详细研究。EFlags 是一个名副其实的垃圾抽屉，里面存放着一些拆散的位信息，但是，如果立刻直入主题，对它们进行详细描述，会很困难(也可能误导)。相反地，我将在这里简单描述一下 CPU 的标志，然后在本章和后面的章节中，在讨论各种各样使用它们的指令的过程中，遇到它们时再进行更为详细的讲解。

一个标志是一个单独的二进制信息位，它的含义独立于任何其他位。根据 CPU 的需要，一个二进制位可以被 CPU 设置为 1 或者清除为 0。这样做是为了告诉你(程序员)CPU 内部的某些条件的状态，以便程序可以对其进行测试并根据那些条件的状态采取相应的行动。更少见的情况是，你(程序员)也可以设置一个标记，并将其作为一种向 CPU 发送某些信号的方式。

考虑一排乡村邮箱，每个邮箱旁边都有一杆自己的小红旗。每面旗帜都可以升起和降下；如果史密斯的旗帜升起，它告诉邮递员说，史密斯已经把邮件放入了自己的邮箱，等待他去拿。邮递员查看(测试)史密斯的旗帜是否升起，如果升起，就打开史密斯的邮箱，取出等待邮寄的邮件。

作为整体来看，EFlags 只是一个掩藏在 CPU 内部的 32 位寄存器。它是出现在 8086/8088 CPU 中的 16 位标志寄存器的 32 位扩展后代。其 32 位中的每一位都是一个标志，尽管有些标志很常见，但是当你刚刚开始学习汇编时，只有较少标志是有用的。此外，还有许多标记没有被英特尔定义，而且尚未使用或暂未使用。

它有点乱，还是看一下图 7-2 吧，它总结了所有当前 x86 体系结构中定义的标志。那些背景变成灰色的标志是晦涩难解的，你目前可以安全地忽略它们。



0	CF	进位标志	0=操作中无进位；1=有进位
1	–	未定义	
2	PF	奇偶标志	0=字节中为1的二进制位的个数是奇数个；1=偶数个
3	–	未定义	
4	AF	辅助进位标志	0=BCD码操作中没有进位；1=BCD码操作中有进位
5	–	未定义	
6	ZF	零标志	0=操作数变为非零值；1=操作数变为0
7	SF	符号标志	0=操作数没有变为负数；1=操作数变为负数
8	TF	单步标志	帮助单步调试
9	IF	中断使能标志	在保护模式下为操作系统预留
10	DF	方向标志	0=自动增长是上内存的；1=自动增长是下内存的
11	OF	溢出标志	0=在有符号操作中没有溢出；1=在有符号操作中有溢出
12	IOPL	0级I/0特权标志	在保护模式下为操作系统预留
13	IOPL	1级I/0特权标志	在保护模式下为操作系统预留
14	NT	嵌套任务标志	在保护模式下为操作系统预留
15	–	未定义	
16	RF	恢复标志	帮助单步调试
17	VM	虚拟8086模式标志	在保护模式下为操作系统预留
18	AC	对齐检查标志	在保护模式下为操作系统预留
19	VIF	虚拟中断标志	在保护模式下为操作系统预留
20	VIP	虚拟中断挂起标志	在保护模式下为操作系统预留
21	ID	CPU识别标志	如果这个位能够被用户空间程序修改的话，CPUID是可用的
22	–	未定义	
		. . .	

图 7-2　x86 的 EFlags 寄存器

EFlags 寄存器中的每一个标志都有一个由两个或三个字母组成的符号，大多数程序员通过这些符号来了解它们。在本书中会用到那些符号，所以你应该熟悉它们。那些最常见的标志的符号及其代表的含义的简要说明如下：

- **OF**：当对一个有符号整数量的算术运算操作结果过于庞大，以至于不能适合原来的操作数时，溢出标志 OF 被设置。当“进位标志”被标记时，溢出标志通常也被设置。
- **DF**：方向标志是几个古怪的标志之一，它告诉 CPU 某些你想让它知道的东西，而不是反过来(CPU 想让你知道)的其他方式。它决定了字符串指令执行过程中的行为移动方向(上内存或者下内存)。当 DF 被设置时，字符串指令从高内存向低内存进行。当 DF 被清除时，字符串指令从低内存向高内存进行。
- **IF**：中断使能标志是一个双向标志。CPU 在某些条件下设置它，你也可以使用 STI 和 CLI 指令来自己设置它。当 IF 被设置时，中断被启用，并且当需要时可以发生。当 IF 被清除时，中断将被 CPU 忽略。早在 DOS 时代，普通程序就可以在实模式下自由地设置和清除这个标志。但在 Linux 下，IF 是供操作系统使用的，有时是为它的驱动程序所用。如果你尝试在自己的程序中使用 STI 和 CLI 指令之一，Linux 将会提交一个一般保护错误，并终止程序。一定要考虑 IF 的“军事禁区”。
- **TF**：当被设置时，单步标志允许调试器通过强制 CPU 在调用一个中断程序前只执行一条指令来实现单步管理。对于一般编程而言，这不是一个特别有用的标志，我也不会对其做更多的讲解。
- **SF**：当一个操作的结果强迫操作数变为负数时，符号标记被设置，“负”只是意味着在一次有符号算术运算中，操作数中排序最高的位(符号位)变成了 1。任何结果为正的算术运算操作都会将 SF 清零。
- **ZF**：当一个操作的结果为零时，零标志被设置。反之，如果目标操作数变成某些非零值，则 ZF 被清零。该标志经常用于条件转移指令中。
- **AF**：辅助进位标志只用于 BCD 码算术运算。BCD 码算术将每个操作数的一个字节看作一对 4 位的“半字节”，并通过使用 BCD 算术指令，允许一些近似十进制(基数为 10)的算术运算直接在 CPU 的硬件上完成。这些指令已经不再使用，本书对 BCD 码算术只进行一些简单讨论。
- **PF**：对于任何了解串行数据通信的人而言，奇偶标志看起来应该很熟悉，但是对于那些不熟悉的人们而言，则完全是件完全奇怪的事情。PF

表明某个结果数字的排序最低的字节中被设置为 1 的二进制位的个数是奇数还是偶数。例如，如果结果为 0F2H，那么 PF 将被清除，因为 0F2H(11110010)中包含奇数个设置为 1 的二进制位。同样，如果结果是 3AH(00111100)，那么 PF 将被设置，因为结果包含偶数(4)个设置为 1 的二进制位。该标志是计算机串口通信遗留下来的。对于串口通信而言，名为奇偶校验的差错检测系统需要依赖于知晓一个字符字节中已被设置为 1 的二进制位的个数是奇数还是偶数。PF 很少使用，本书将不再对其进行进一步说明。

- **CF**：进位标志用于无符号算术运算。如果一个算术或移位操作的结果从操作数“进”1 位，CF 被设置。否则，如果没有进位，CF 将被清除。

7.2.1　标志规范

我所说的“标志规范”是指给定的指令影响 EFlags 寄存器中的标志的方式。必须记住，刚才所讲的标志描述只是概括，受特定的限制及个别指令限制的特殊情况所支配。针对单个标志的规范在不同的指令中变化很大，尽管标志的使用应该是在任何情况下都相同。

例如，某些导致操作数中出现零的指令会设置 ZF，而另一些指令则不会。很遗憾的是，没有针对它的系统，也没有简单的方法来将其直接印在脑海中。当你打算使用条件转移指令的方式来测试标志位时，必须检查每条单独的指令，以观察各种标志如何受到影响。

标志规范是一件非常个体化的事情。请查看每一条指令的指令参考部分，看看它是否影响了标志。不要想当然！

关于标志规范方面的一节简单的课程涉及两条指令：INC 和 DEC。

7.2.2　使用 INC 指令和 DEC 指令加 1 和减 1

有些 x86 机器指令是成对出现的。这些指令之中最简单的就是 INC 和 DEC，它们将一个操作数分别加 1 和减 1。

将某些内容上加 1 或减 1，这是在计算机编程中经常发生的操作。如果要对一个正在执行循环的程序所执行的循环次数进行计数，或者对一张表中的字节数进行计数，或者做某些一次前进一步或后退一步的事情，INC 和 DEC 可能是使加法和减法实际发生的最快的方法。

INC 和 DEC 这两个指令均只需一个操作数。如果尝试为任意其中一个使用两个操作数，或者根本不使用操作数，汇编编译器将发出一个错误。

通过将以下指令加入到你的沙盒中来尝试使用这两条指令。像往常一样生成沙盒，加载可执行文件到 Insight 中，并且单步通过该沙盒：

```
mov eax,0FFFFFFFFh
mov ebx,02Dh
dec ebx
inc eax
```

观察 EAX 和 EBX 寄存器发生了什么。可以预见，递减 EBX 会将值 2DH 变成 2CH。另一方面，递增 0FFFFFFFFH，会将 EAX 寄存器的值卷为 0，因为 0FFFFFFFFH 是能够用一个 32 位寄存器表示的最大无符号值。该值与某个数相加将会使其回滚为零，就像加 1 到 99 上会使得总和的最右边两个数字变为零，进而创造了数字 100 一样。INC 的不同之处是它没有进位。进位标志不受 INC 指令的影响，所以不要尝试用它来执行多位数算术运算。

7.2.3 从 Insight 中观察标志

像 EAX 一样，Eflags 寄存器是一个寄存器，它的值在 Insight 的寄存器视图中更新。不幸的是，在该视图中看不到单独的标志。相反地，它给出的是整个 Eflags 中的十六进制值，就像每个标志位都是这个十六进制数中的一位一样。如果你在观察一条指令对某个特定标志所做的事情，这就不是特别有用了。执行上述的 DEC EBX 指令，将 Eflags 中的值从 0292h 修改为 0202h。在我们的标志中发生了一些改变，但是，是那些发生了改变呢？

这的确是一个 Insight 调试接口不擅长的地方，特别是对于那些首次探索 x86 指令集的人而言。没有简单的视图来展示单独的 Eflags 中的标志值。要想单独了解哪些标志发生更改，必须打开控制台视图并在控制台命令行中执行 Gdb 命令。

从 Insight 主菜单中选择 View(视图)→Console(控制台)。控制台视图很简单：只有一个空白终端，右上角带有提示符(gdb)。单步运行沙盒到 DEC EBX 指令，但是在执行该指令之前，在控制台视图键入如下命令：

```
info reg
```

你所看到的应该如图 7-3 所示。Gdb 的 info 命令显示某些内容的当前状态，而 reg 参数则告诉 Gdb，我们想要查看寄存器的当前状态。当然，你将得到不止

Eflags 一个寄存器，默认情况下，所有的通用寄存器和段寄存器都将被显示出来。数学处理器寄存器不会显示，这样刚好有利于我们的操作目的。

看看图 7-3 中显示 Eflags 值的那一行。寄存器的十六进制值被整个显示出来，但是在这个值之后的方括号内是一个寄存器名称列表。里面只显示那些被设置的标志，而那些被清除的标志则不显示。在执行 DEC EBX 指令之前，AF、SF 和 IF 标志已经被设置。现在，执行 DEC EBX 指令。然后再次在控制台视图中输入 info reg 命令。显示 EFlags 值的那一行已经被改成了这样：

```
eflags 0x202 [IF]
```

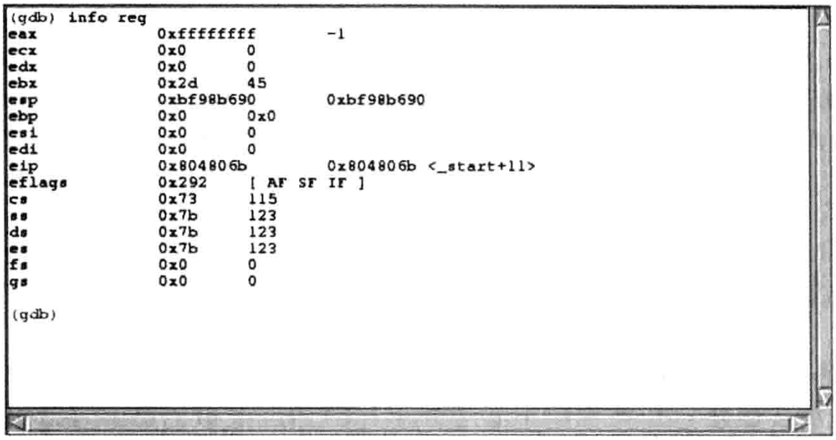


图 7-3　在 Insight 的控制台视图中显示寄存器

那么，到底发生了什么事情？看一下附录 A 中有关 DEC 指令的那一页，你也许能得到一些提示：DEC 影响 OF、SF、ZF、AF 和 PF 标志。DEC EBX 指令将所有这些标志的值清零。原因如下：

- 溢出标志(OF)被清除，因为被解释为有符号整数的操作数并没有变得太大，以至于无法存放在 EBX 中。如果不知道是什么使一个数变为有符号数的话，这可能对你没有帮助，所以暂时先不管它。
- 符号标志(SF)被清除，因为操作的结果没有使得 EBX 的最高位变成 1。假如 EBX 的最高位变成 1，那么，被解释为有符号整数的 EBX 的值将成为负数，当一个值变为负数时，SF 被设置。所以，像 OF 一样，SF 也没有多大用处，除非正在做有符号算术运算。
- 零标志(ZF)被清零，因为目标操作数不等于零。如果为零，ZF 将被设置为 1。

- 辅助进位标志(AF)被清零，因为没有从 EBX 的低四位向高四位的 BCD 码进位。
- 奇偶标志(PF)被清零，因为在递减操作发生后，操作数中的二进制位 1 的数量为 3，当目标操作数中的二进制位 1 的数量是奇数时，PF 被清零。请自己检查一下：在 DEC 指令执行后， EBX 中的值变为 02Ch。以二进制表示，就是 00101100。在这个值里面有 3 个值为 1 的二进制位，因此 PF 被清除。
- DEC 指令并不影响 IF 标志，它仍然被设置为 1。事实上，几乎没有什么指令能够改变 IF 标志，用户模式的应用程序(比如沙箱和一切你在学习汇编过程中可能编写的程序)禁止修改 IF。

现在，执行 INC EAX 指令，并重新在控制台视图显示寄存器的值。轰！这次有许多行动发生：

- 奇偶标志(PF)被设置，因为 EAX 中出现的值为 1 的二进制位的个数是零，当操作数的值为 1 的二进制位的个数是偶数时，PF 被设置为 1。0 被认为是偶数。
- 辅助进位标志(AF)被设置，因为 EAX 的低 4 位从 FFFF 变成了 0000。这意味着从低 4 位到高 4 位有进位，当操作数的低 4 位有进位时，AF 被设置。
- 零标志(ZF)被设置，因为 EAX 变成了零。
- 像前面一样，IF 标志没有变化，仍然一直处于设置状态。

7.2.4 标志如何改变程序的执行

观察指令执行后各个标志值发生的改变，这是一个学习标志规范的好办法，但是一旦已经理解了各种指令如何改变标志，就可以关闭控制台视图了。标志的真正价值不在于它们的值本身，而是在于它们如何影响机器指令在程序中的流动。

有一类机器指令，它们根据某个标志的当前值，跳转到程序中的不同位置。这些指令被称为条件转移指令，Eflags 中的大部分标志都有一个或者多个相关的条件转移指令。附录 A 列出了它们。

回想一下第 1 章中介绍的“步骤”和“测试”的概念。多数机器指令是一个采取的步骤的列表，一般从上往下运行。条件转移指令是测试。它们测试某个标志的值，要么继续往下走，要么跳转到程序的其他部分。

最简单而且可能使用最多的条件转移指令是 JNZ，如果不为零则跳转。JNZ 指令测试零标志的值。如果 ZF 已被设置(即等于 1)，则什么也不发生，CPU 顺序执行下一条指令。但是，如果 ZF 没有被设置(即等于 0)，那么执行将转到程序中的一个新目的地“去旅游”。

实际情况没有听起来那么糟糕。你不用担心加或者减任何东西。几乎在所有情况下目标位置都提供一个标号。标号是赋给程序中某个位置的描述性名称。在 NASM 中，标号是一个字符串，后面跟一个冒号，一般放在包含指令的代码行上。

像汇编语言中的许多情况一样，简单的例子会让知识变得更加清晰。加载一个新的沙盒，然后敲入以下指令：

```
        mov eax,5
DoMore: dec eax
        jnz DoMore
```

生成沙盒程序并将其加载到 Insight 中。在单步执行该程序的过程中，通过寄存器视图观察 EAX 中的值。尤其要观察当 JNZ 指令执行时，源代码窗口会发生什么。如果 ZF 为 0，JNZ 跳转到以其操作数命名的标号处。如果 ZF= 1，它则“失败”地进入下一条指令。

DEC 指令递减 EAX 中的值。只要 EAX 中的值不为 0，零标志将一直保持清除状态。并且只要零标志被清除，JNZ 就跳回到标号 DoMore。因此，对于这五次循环，每次 DEC 都将 EAX 的值减少 1，JNZ 跳回到 DoMore。但是，只要 DEC 把 EAX 的值减少到 0，零标志就被设置，JNZ“失败地”进入沙盒结尾处的“NOP”指令。

这样的结构叫做*循环*，它在所有的编程中都非常普遍，不仅仅在汇编语言编程中。前面的循环没有什么实际用处，但它演示了如何将一条指令重复执行需要的次数，即通过将一个初始值加载到寄存器中，然后每循环一次，将那个值递减。JNZ 指令在每一次循环中都测试 ZF，并且知道当计数器变为 0 时停止循环。

我们可以在不增加多少复杂性的情况下让这个循环变得有用些。我们急需添加的是一个用于循环工作的数据项。将程序清单 7-2 加载到 Kate 编辑器，生成它，然后将其加载到 Insight 中。

清单 7-2：ngaroo.asm

```
section .data
        Snippet db "KANGAROO"
```

```
section .text
global _start
_start:
        nop
; 将你的实验放在两个 nop 指令之间......
        mov ebx,Snippet
        mov eax,8
DoMore: add byte [ebx],32
        inc ebx
        dec eax
        jnz DoMore

; 将你的实验放在两个 nop 指令之间......
       Nop
```

与一般沙盒程序的唯一区别是变量 Snippet 和 nop 之间的 6 条指令。确保 Insight 的内存视图处于打开状态，然后单步执行程序。

经过 8 次循环之后，KANGAROO 已经变成了 kangaroo。这是如何实现的呢？看看位于标号 DoMore 处的 ADD 指令。在程序的前面部分，我们将 Snippet 的地址复制到寄存器 EBX 中。ADD 指令把字面常量 32 加到任何以存放在 EBX 中的数字为地址的数据上。如果查看附录 B，你会发现，大写字母的 ASCII 值和小写字母的 ASCII 值之间的差值为 32。大写字母“K”的值为 4Bh，小写字母“k”的值为 6Bh。6Bh- 4Bh 是 20h，也就是十进制的 32，因此，如果将 ASCII 字母看成数字的话，可以通过将大写字母(的 ASCII 码值)加上 32 来将其转化为小写字母。

循环执行了 8 次，每一次用于转换 KANGAROO 中的一个字母。在每个 ADD 指令执行之后，程序都会递增 EBX 中的地址，这样就将“KANGAROO”中的下一个字符放到“瞄准镜”下。同时它还递减 EAX，在循环开始之前，EAX 中已经装载了变量 Snippet 中存放的需要处理的字符数。因此，在同一个循环中，程序对于存放在 EBX 中的 Snippet 的长度进行累加计数，而对于存放在 EAX 中的值进行递减计数。当 EAX 到达零时，就意味着处理完了 Snippet 中的所有字符，任务完成。

ADD 指令的操作数值得我们密切关注。把 EBX 放到方括号内，引用的是 Snippet 的内容，而不是它的地址。但更重要的是，byte 大小说明符告诉 NASM，我们只是正在将一个字节写到 EBX 中存放的内存地址中。否则 NASM

没有办法知道写入多少内容。它可能一次写一个字节、两个字节或者四个字节到内存，取决于需要做的事情。但是，必须告诉 NASM 你想让它做什么。

不要忘记，kangaroo.asm 仍然只是一个沙盒程序，它只适合在调试器中进行单步调试。如果让它运行，那么，当执行超出最后的 NOP 指令时，它将产生一个分段错误。一旦单步调试到最后的 NOP 指令，kill 该程序，然后要么重新开始执行，要么退出调试器。

7.3 有符号值和无符号值

在汇编语言中，既可以使用有符号数，也可以使用无符号数。当然，有符号数是可以为负数的值，而无符号数的值则始终为正。在 x86 指令集中，有一些用于四种基本算术运算的指令，这些指令既可以操作有符号值，也可以操作无符号值。(对于乘法和除法，有单独的指令分别用于有符号和无符号计算，我会在稍后进行介绍。)

理解有符号数和无符号数之间区别的关键是知道 CPU 将符号放在什么地方。它不是一个破折号符号，而是实际上代表这个数字的二进制模式下的一个位。一个有符号数的最高有效字节的最高位是符号位。如果符号位的值为 1，则这个数为负数。如果符号位的值为 0，则这个数为正数。

记住，一个给定的二进制模式到底代表一个有符号数还是无符号数，取决于你选择如何使用它。如果打算执行有符号算术运算，一个寄存器值的高位或者内存位置的高位被认为是符号位。如果不打算执行有符号算术运算，那么同样的值的同样位置的高位则简单地代表无符号数的最高有效字节中的一位。一个值的符号的本质在于如何对待这个值，而不在于代表该值的底层位模式的本质。

举例来说，二进制数 10101111 代表有符号值还是无符号值呢？如果没有上下文的话，这样的问题毫无意义：如果需要将其作为有符号值，就把排序最高的那个位作为符号位，那么这个值为-81。如果需要将其作为无符号值，只需将这个高位看成该二进制数字的一个位即可，那么这个值为 175。

7.3.1 补码和 NEG

初学者有时会犯的一个错误是假设可以通过将符号位设置为 1 来将一个值变为负数。而事实情况绝非如此！不能简单地拿出 42 这个值，并通过将符号位设置为 1 来将其变为-42。得到的值肯定是负的，但它不会是-42。

了解负数在汇编语言中的表达方式的一种途径就是将一个正数递减，一直减到负数区域。建立一个干净的沙盒，输入这些指令：

```
    mov eax,5
DoMore: dec eax
    Jmp DoMore
```

像往常一样生成沙盒，并将可执行文件加载到 Insight 中。请注意，这里我们已经添加了一条新的指令，同时也添加了一个危险：JMP 指令并不查看标志。当执行时，它总是跳转到它的操作数，进而跳转到助记符。因此，每一次 JMP 指令执行时，执行都将回退到标号 DoMore。如果足够敏锐的话，你会发现，根本没有跳出这种特殊指令序列的路，是的，这就是你将偶尔遇到的传说中的“无穷循环”(死循环)。

因此，一定要确保在 MOV 指令上设置处初始断点，而不只是让程序飞速运行。或……干脆让程序继续前进！(没有什么东西将受到伤害。) 如果没有断点，你将看到，Insight 的“跑步人”图标变成一个停止的符号。而当你看到停止标志的图标时，就会知道程序不是暂停用于单步调试，而是正在自由运行。如果点击停止标志，Insight 将停止执行该程序。在 DOS 下，你会被卡住，不得不重新启动。Linux 和 Gdb 准备了一个更为强大的编程环境，一个不会轻易被少量错误打败的环境。

开始单步调试这个“沙盒”，并在寄存器视图中观察 EAX。起始值 5 将会被减小到 4、3、2、1 和 0，然后……0FFFFFFFFh！这是以 32 位表达的简单值-1。如果将 EAX 继续递减下去，你会得到如下情况：

```
0FFFFFFFFh (-1)
0FFFFFFFEh (-2)
0FFFFFFFDh (-3)
0FFFFFFFCh (-4)
0FFFFFFFBh (-5)
0FFFFFFFAh (-6)
0FFFFFFF9h (-7)
```

……等等。当负数以这种方式处理时，它就称为补码。在 x86 汇编语言中，负数存储为其绝对值的补码形式，它是一个数字与 0 之间的距离，无论是在正数方向，还是在负数方向，如果你还记得 8 年级数学的话。

补码背后的数学更是令人惊讶的微妙，我建议到维基百科上获取对它更为全面的了解：

http://en.wikipedia.org/wiki/Two's_complement

以补码形式表达负数的魔力是：CPU 在晶体管逻辑层面上，并不真正需要减法。它只是产生减数的补码，并将其与被减数相加。对于 CPU 而言，这是比较容易的，而且发生的一切对程序都是透明的，减法被按照预期的方式完成。

好消息是，你几乎从来不必手动计算一个值的补码。机器指令 NEG 会为你完成这件事。该指令将一个正数作为它的操作数，然后将这个值取负，也就是说，使它成为负数。NEG 通过生成这个正数的补码形式来实现操作。将以下指令加载到一个干净的沙盒中，并在 Insight 中对其进行单步调试。在寄存器视图中观察 EAX：

```
mov eax,42
neg eax
add eax,42
```

一下子，42 变成了 0FFFFFFD6h，-42 的补码的十六进制表达形式。将 42 与这个值相加，我们会看到 EAX 变为了 0。

此时，问题就可能产生了：用一个、两个或者四个字节能表示的最大正数和最大负数是多少呢？这两个值，再加上这两个值之间的所有值就构成了给定数量的位能够表示的值的范围。我已将这个内容展示在表 7-2 中。

表 7-2　有符号数的范围

值的位数	能表示的最大负数		能表示的最大正数	
	十进制	十六进制	十进制	十六进制
8 位	-128	80h	127	7Fh
16 位	-32768	8000h	32767	7FFFh
32 位	-2147483648	80000000h	2147483647	7FFFFFFFh

如果你很敏锐，并且知道如何在十六进制中计数，你可能会从这里的表中注意到一些东西：对于一个给定的位数而言，它所能表示的最大正值与最大负值之间相差 1。也就是说，如果处理 8 位的值，将 8 位最大正数的值(7Fh)加 1，将得到一个最大的负值(80h)。

可以通过在沙盒中执行以下两条指令来观察这种情况：

```
mov eax,07FFFFFFh
inc eax
```

这个例子连同我之前向你展示的技巧一起才有意义：Insight 的寄存器视图允许以三种不同的形式显示寄存器的值。默认情况下，Insight 以十六进制显示所有寄存器的值。但你可以在寄存器视图窗口右击任意一个给出的寄存器，然后在十六进制，十进制和无符号数之间进行选择。这三种格式采用如下方式工作：

- 十六进形式以十六进制表示该值。
- 十进制形式将该值表示为一个有符号数，把最高位当成符号位来处理。
- 无符号形式将该值表示为一个无符号值，把最高位只是看成整个数中的一个二进制位。

所以在执行上述两个指令之前，在寄存器视图中，右击 EAX 寄存器并选择十进制。执行 MOV 指令后，EAX 将显示十进制值 2147483647。这是 32 位能表示的最大有符号值。通过 INC 指令递增这个值，EAX 中的值立即变成了-2147483648。

7.3.2 符号扩展和 MOVSX

当处理不同位数的有符号数时，要避免一个令人费解的“疑难杂症”。符号位是一个有符号字节、有符号字或者有符号双字的最高位。但是，当必须将一个有符号值移到更大的寄存器中或者内存中时，会发生什么情况呢？例如，如果需要将一个 16 位有符号值移到 32 位寄存器中，将会发生什么？如果使用的是 MOV 指令，那么什么用处也没有。试试如下指令：

```
mov ax,-42
mov ebx,eax
```

-42 的十六进制形式是 0FFD6h。如果将其存放在一个 16 位寄存器中，比如 AX，并且使用 MOV 指令将其移到一个 32 位寄存器中，比如 EBX，那么符号位将不再是符号位。换句话说，一旦-42 从一个 16 位容器到一个 32 位的容器中去“旅行”，它将从-42 变为 65494。符号位仍然存在。它也没有被清零。然而，在一个 32 位寄存器中，原来的符号位现在只是一个二进制值中的普通位，根本没有什么特殊含义。

这个例子有点误导。首先，我们不能照字面意义将一个值从 AX 中移到 EBX 中。MOV 指令只负责处理位数相同的操作数。但是，请记住，AX 只是 EAX 寄存器中的两个较低字节。我们可以通过把 EAX 中的值移到 EBX 中来实现将 AX 中的值移动到 EBX 中，这就是前述示例所做的事情。

可惜，Insight 不能向我们显示有符号 8 位值或 16 位值。Insight 只能显示 EAX，我们只能在 EAX 内部看到 AL、AH 或者 AX。这就是为什么在前面的例子中，Insight 将我们认为是-42 的值显示成了 65494 的原因。Insight 寄存器视图并没有符号位的概念，除非该位是一个 32 位值的最高位。这是 Insight 程序本身的一个缺点，我希望有人能够最终改善寄存器视图，以便让有符号 8 位值和 16 位值能够这样显示。

x86 CPU 提供了一种能够跳出这个陷阱的方法，那就是 MOVSX 指令。MOVSX 的意思是“带符号扩展一起移动”，它也是在原来 8088/8086 CPU 中不存在的诸多指令之一。MOVSX 是随着 386 系列的 CPU 而引入的，因为 Linux 无法在比 386 古老的计算机上运行，所以可以假设，任意的 Linux 个人电脑都支持 MOVSX 指令。

将以下内容加载到沙盒中，并尝试执行它：

```
mov ax,-42
movsx ebx,ax
```

记住，Insight 不能单独显示 AX，因此会将 EAX 显示为包含数值 65494。但是，当通过 MOVSX 指令将 AX 移到 EBX 中时，EBX 中的值将被显示为-42。所发生的事情是，MOVSX 指令对它的操作数执行了有符号扩展，把符号位从 AX 中的 16 位量中拿出来，使其成为位于 EBX 中的 32 位量的符号。

MOVSX 指令与 MOV 指令不同之处在于，它的操作数可以具有不同的位数。MOVSX 有三种可能的变体，我已经将其总结在表 7-3 中。

表 7-3　MOVSX 指令

机器指令	目标操作数	源操作数	操作说明
MOVSX	r16	r/m8	8 位有符号数到 16 位有符号数
MOVSX	r32	r/m8	8 位有符号数到 32 位有符号数
MOVSX	r32	r/m16	16 位有符号数到 32 位有符号数

注意，目标操作数只能有一个寄存器。在描述指令操作数方面，很多汇编语言参考资料中都使用这里的符号。符号“r16”是一个任意的 16 位寄存器的缩写。同样，“r/m”的意思是任意的寄存器或者内存位置，后面跟它们的位数。例如，“r/m16”是指任意的 16 位寄存器或内存位置。

上述内容介绍得很全面，但在解决了一些汇编语言问题之后，你可能会发

现，有符号算术运算的应用远比想象中的少得多。了解它的工作原理是有好处的，但是，如果你已经使用汇编语言工作了很多年却从未使用过它的话，千万不要感到惊讶。

7.4　隐式操作数和 Mul

大多数时候，我们通过一个或两个(位于某行代码中的助记符旁边的)操作数将值提交给机器指令。这样很好，因为谈到 `MOV EAX, EBX` 的时候，我们确切地知道正在移动什么、数据来自何方以及要到哪里去。不过，情况并非总是如此。有些指令操作寄存器甚至操作不能用操作数列表指出的内存位置。这些指令实际上也有操作数，但是这些操作数代表的是由指令做出的假设。这种操作数被称为隐式操作数，它们永远不变，也不能被更改。使情况变得更加复杂的是，大部分包含隐式操作数的指令同时也包含显式操作数。

在 x86 指令集中，最好的隐式操作的例子是乘法和除法指令。排除专门的数学处理器(x87、MMX 和 SSE，我不会在本书中涵盖它们)中的指令，x86 指令集有两套乘法和除法指令。一组是 MUL 和 DIV，用于处理无符号计算。另一组是 IMUL 和 IDIV，用于处理有符号计算。因为 MUL 和 DIV 比它们对应的有符号运算指令使用频繁，因此它们是我在本节讨论的对象。

MUL 指令完成我们期望它所做的事情：它将两个值相乘，然后返回一个结果。然而，在基本的数学运算中，乘法有一个特殊的问题：它产生的输出值经常比输入值大很多。这使得我们无法按照常规的 x86 指令模式操作，即把指令产生的结果值存入目标操作数中。

考虑一个 32 位乘法操作。32 位寄存器中能够存放的最大无符号值是 294 967 295。它们即使是与 2 相乘也会得到一个 33 位的结果，而该结果不再适合于任何 32 位寄存器。这个问题从一开始就一直困扰着 x86 体系机构(实际上，所有计算机体系结构)。当 x86 是 16 位体系结构时，就存在着将两个 16 位值的、非常容易溢出 16 位寄存器的计算结果存放到何处的问题。

为了解决这个问题，Intel 的设计师们采用他们唯一能够做到的方法：通过使用两个寄存器来保存计算结果。对于非数学家而言，也许不是显而易见的，但是两个二进制数的最大乘积的位数不超过较大因子的位数的两倍(在计算器上试一下吧！)这是事实。简单地说，任意两个 16 位值的乘积都将适合存储在 32 位容器中，而任意两个 32 位值的乘积都将适合存储在 64 位容器中。因此，当需要两

个寄存器来存放乘积的时候，就永远只需要两个寄存器。

这就给我们带来了 MUL 指令。从操作数的角度来看，乘法指令是一只怪鸟：它只采用一个操作数，该操作包含一个即将相乘的因子。另外一个因子是隐式的，它就是那对接收计算结果的寄存器。MUL 指令因此看起来好像很简单：

```
mul ebx
```

这里不仅仅涉及 EBX 寄存器。隐式操作数取决于显式操作数的位数。这带给我们 3 种变体积，我已经在表 7-4 中进行了总结。

表 7-4　MUL 指令

机器指令	显式操作数(因子 1)	隐式操作数(因子 2)	隐式操作数(结果)
mul	r/m8	AL	AX
mul	r/m16	AX	DX and AX
mul	r/m32	EAX	EDX and EAX

第一个因子以一个明确的操作数给出，它可以是一个寄存器中的值，也可以是某个内存位置中的值。第二个因子是隐式的，并且总是在适合于第一个因子位数的“A”系列通用寄存器中。如果第一个因子是一个 8 位值，第二个因子总是在 8 位寄存器 AL 中。如果第一个因子是一个 16 位值，第二个因子总是在 16 位寄存器 AX 中，以此类推。

一旦结果需要不止 16 位，“D”系列寄存器就被“征用”，以存放计算结果的高序位部分。这里的“高序位”指的是不适合在“A”系列寄存器中存放的那部分计算结果。例如，如果将两个 16 位值相乘，得到结果为 02A456Fh，那么，寄存器 AX 中会包含 0456Fh，而 DX 寄存器中则存放 02Ah。

一定要注意，即使计算结果足够小，以至于存放结果的两个寄存器之一就能够存放它，高序位寄存器(不论是 AH，DX，或 EDX)仍然被清零。在汇编工作中，寄存器经常变得越来越稀少，但是，即使能够确保乘法总是包含较小的结果，在乘法指令执行过程中，也不能将高序位寄存器用于除 MUL 之外的任何其他指令中。

同时，也要注意， MUL 指令的操作数不能为立即数；也就是说，你不能做以下事情，将第一个操作因子声明为立即数，那样它将不再像往常那样有用：

```
mul 42
```

7.4.1 MUL 和进位标志

并非所有的乘法运算都能够产生足够大的、需要两个寄存器存放的计算结果。在大多数情况下你会发现，32 位就绰绰有余了。那么，如何知道在高序位寄存器中是否存在有效数字呢？当乘法结果的值溢出低序位寄存器时，乘法指令将设置进位标志(CF)，这样就给我们带来了很大的帮助。经过一次 MUL 指令后，如果发现 CF 被设置为 0，则可以忽略高序位寄存器，非常放心地认为整个计算结果存放在两个寄存器中的低序位寄存器中。

这一点很值得采用一个快速的沙盒来演示一下。首先，尝试一个较“小”的乘法，该乘法的结果很容易在一个 32 位寄存器中存放：

```
mov eax,447
mov ebx,1739
mul ebx
```

记住，这里是在将 EBX 与 EAX 相乘。单步通过这三条指令，当乘法指令执行后，在寄存器视图中观察 DEX 和 EAX 中的结果。EAX 包含 777333，EDX 中包含 0。现在，在控制台视图中键入 info reg，并查看各种标志的当前状态。没有 CF 标志，也就是说，CF 已被清 0。

接下来，将下面的指令添加到沙盒程序中，放在上述示例中的那三条指令后面：

```
mov eax,0FFFFFFFFh
mov ebx,03B72h
mul ebx
```

像以往一样单步通过这些指令，在寄存器视图中观察 EAX、EDX 和 EBX 中的内容。在 MUL 指令执行后，再次在控制台视图中键入 info reg。进位标志(CF)已被设置为 1。(所以会有溢出标志、OF、符号标志、SF、奇偶标志、PF，但是这些在无符号算术中一般都没有用处。)这里，CF 主要告诉你的是在计算结果的高位部分存在有效数字，而这些有效数字存放在用于 32 位乘法运算的 EDX 寄存器中。

7.4.2 使用 DIV 实现无符号除法

记得还是一名小学三年级学生时，我曾经在班上断然指出，除法就是反着做

乘法，我比可怜的艾格尼丝·艾琳修女那时愿意接受的事实更加接近真相。在 x86 无符号乘法指令 MUL 和无符号除法指令 DIV 之间，的确存在着很强的相似性，这一点肯定是正确的。DIV 所做的就是小学三年级训练所期望的事情：它将一个值被另外一个值除，然后得出一个商和一个余数。记住，这里我们正在进行整数运算，而不是小数运算，所以没有办法来表达诸如 17.76 或 3.14159 之类的十进制商数。如果需要执行小数运算，需要 x86 体系结构中的数学处理器的“浮点”机制，它是一个庞大而微妙的问题，我不会在本书中涵盖。

在除法运算中，不存在乘法运算所具有的对于某些输入值产生较大的输出值的问题。如果以一个 16 位值除另一个 16 位值，永远不会得到一个不适合存放于 16 位寄存器中的商。然而，能够除以一个非常大的数，这是有益的，所以 Intel 的工程师们创造了很像 MUL 镜像的东西：你把被除数放在 EDX 和 EAX 中，这意味着它的大小最大可达 64 位。64 位可以容纳非常大的数：18 446 744 073 709 551 615。除数被存放在 DIV 的唯一显式操作数中，它可能在寄存器中或者在内存中。(同乘法指令一样，不能使用立即数作为操作数。)返回的商数存放在 EAX 中，余数存放在 EDX 中。

这就是一个 32 位除法的全面情况。如同乘法指令一样，DIV 的隐式操作数取决于显式操作数的尺寸，因此在这里作为除数来使用。有三种“尺寸”的 DIV 操作，如表 7-5 中总结的内容所示。

表 7-5 DIV(除法)指令

机器指令	显式操作数 除数	隐式操作数 商的整数部分	隐式操作数 商的余数部分
DIV	r/m8	AL	AH
DIV	r/m16	AX	DX
DIV	r/m32	EAX	EDX

DIV 指令不影响任何标志。然而，除法的确存在一个特殊的问题：无论被除数还是除数，只要值为 0 就被认为是未定义的，这就会产生一个 Linux 算术异常，该异常将终止你的程序。所以，在执行 DIV 之前先测试被除数和除数的值，以确保它们统统不为零，这一点非常重要。

你可能会提出异议，普通的小学数学允许零被一个非零值除，其结果始终为零。这在数学上的确是正确的，但它不是一个特别有用的操作，在 x86 体系结构中，零被任何值除都将导致错误。

本书的后面部分将演示一个有用的 DIV 指令的应用，我们将建立一个程序，用于将一个纯粹的二进制值转换为可以在电脑屏幕上显示的 ASCII 字符串。

7.4.3　x86 中的“慢动作”指令

关于 MUL 指令和 DIV 指令，初学者遇到的一个共同问题均涉及这两条指令的两个较小“版本”(见表 7-5)。如果 32 位乘法或者除法可以处理 x86 体系结构的 IA32 实现能够装入寄存器中的任何数据，为什么较小版本甚至是必不可少的呢？难道它仅仅是为了实现与老式的 16 位 CPU 的向后兼容吗？

不完全是。在很多情况下，这是一个速度问题。DIV 和 MUL 指令接近整个 x86 指令集中最慢的指令。当然，它们不会比它们以往的速度慢，但是与其他指令相比，如 MOV 或 ADD，它们简直就是“笨蛋”。此外，两条指令的 32 位版本均慢于 16 位版本，8 位版本是所有版本中最快的。

现在，速度优化是 x86 世界中一项非常棘手的业务。将指令存放在 CPU 的缓存中与将指令存放在内存中相比较，二者的速度差异已经淹没于众多指令本身的速度差异之中。在最新的奔腾级 CPU 指令中，其他因素开始发挥作用，这使得泛泛地谈论指令速度几乎成为不可能的事情，当然更不可能对其进行精确地说明了。

如果只是做一些孤立的乘法或除法运算，不要让其中的速度问题打扰你。但在那些不断执行大量计算的循环中，比如图形绘制和视频处理(如果正在做此类事情，你也许应该使用 x86 体系结构的数学处理器部分，而不是 MUL 指令和 DIV 指令)，指令速度至关重要。我的个人启发是使用输入值允许的最小版本的 MUL 指令和 DIV 指令，而且大多数时候被更大的启发所冲淡，指令速度并不重要。当汇编语言经验足够丰富，以至于能够进行指令级性能决策时，你将知道这一点。在此之前，你应该集中精力在程序错误上，而将速度问题扔给 CPU。

7.5　阅读和使用汇编语言参考资料

汇编语言编程是关于细节的编程。天哪！它居然关乎细节！指令之间有着大量的相似之处，但是，当你开始向汇编编译器那冷酷无情的眼睛提供自己程序时，你得到的恰恰是它们之间的区别。

记忆大量涉及几十种不同指令的、微小的、纠缠的细节是残酷的，也是不必要的。即使是那些汇编高手也不试图一直记忆那些内容。大部分人在手头都预留

一些其他类型的参考文档，以唤起他们对于机器指令细节的记忆。

7.5.1 对于复杂记忆的唤醒文件

这个问题已存在了很长时间。35 年前，当我第一次遇到微型计算机时，一个完整的、有用的指令集记忆唤醒文件可以占用那种能够放在衬衣口袋里的三折卡片的两面。这种卡片非常常见，你可以得到几乎所有微处理器的相关卡片。不知出于什么原因，它们被称为蓝卡，尽管大多数印制在普通的白色纸板上。

在 20 世纪 80 年代初，曾经存放在一张卡片上的内容已经变成了一个 89 页的小册子，大小也适合你的口袋。Intel 程序员的 8086 系列 CPU 袖珍参考指南是为微软的宏汇编而配备的，我所知道的每一个人都有一本。(我的那本至今依然健在。)它确实适合放在衬衣口袋里，前提是口袋中不能装别的东西。

到了 80 年代中期，x86 体系结构的能力和复杂性呈爆炸性增长，所有以它们自己形式的指令的全面总结加上各种必要的解释，构成了书中的材料，并且随着时间的流逝，要想完全覆盖它，已经需要不止一本书，而是若干本书。Intel 提供其处理器文档 PDF 版本的免费下载，可以通过如下网址得到它们：

www.intel.com/products/processor/manuals/

它们是值得拥有的，但是请彻底忘掉将它们塞进口袋的事情。单是指令集参考资料就需要两本 1600 页的书来展示，围绕着指令集的其他必要书籍还有四到五本。

我见过的也许最好的折中办法是 Borland 的 *Turbo Assembler* 快速参考指南。这是一个 5" × 8'的螺旋装订的平折小册子，只有 140 页，作为 Turbo Assembler 产品文档集的一部分，于 1990 年出版。关于汇编指令的材料并不适用于 NASM，但是它的指令参考包括能够在 486 上通过的所有 32 位形式的指令，而这是一个汇编初学者可能使用的几乎所有内容。

Turbo Assembler 快速参考指南的拷贝经常可以在一些在线书籍网站上看到，如 Alibris(*www.alibris.com*) 和 ABE Books(*www.abebooks.com*)，其定价在 5 美元到 10 美元之间。

7.5.2 初学者汇编语言参考指南

关于汇编语言参考资料的问题是，要想让它完整，体积就不能小。然而，现代 x86 中的很多复杂指令和内存寻址机制只为操作系统和驱动程序使用。对于短

小的运行在用户模式下的应用，它们根本不适用。

因此，考虑到那些刚刚开始使用汇编语言的人，我把一些初学者经常参考的、最常见的 x86 指令一起放在附录 A 中。对于我在本书中涵盖的每一条指令，附录 A 中都至少包含一页的内容，再加上一些其他的、每个人都应该知道的指令。它并不包含每一条指令的描述，而是只对那些最常见和最有用的进行描述。一旦你已经足够熟练，可以使用那些更为神秘的指令时，你应该能够阅读英特尔的 x86 文档，并且将其运用在汇编工作中。

附录 A 中有一个简单示例项，在下面的讨论中可以参考它。

指令助记符在页面顶部，带阴影且高亮显示，以便你在迅速翻阅附录时容易找到。助记符的右侧是这条指令的名字，它比光秃秃的助记符多了一点描述。

7.5.3　标志

在助记符的下方是一个存放在 CPU 的 EFlags 寄存器中的标志位的迷你图。如前所述，EFlags 寄存器是一个 1 位值的集合，每个标志保留了短时间内机器状态的基本信息。许多(但并非所有)x86 指令会改变一个或多个标志的值。然后条件转移指令单独测试这些标志，并根据它们的不同值而改变程序的执行流程。

每个标志都有一个名字，每个标志在迷你图中也都有一个符号。随着接触汇编语言的时间越来越长，你最终肯定能够通过这些两字符标志符号知道这些标志，但是直到那时，标志的全名才被显示在迷你图的右侧。大部分标志都不是刚开始学习汇编语言时经常使用的。在迷你图中，你应该关注的大多数标志是零标志(ZF)和进位标志(CF)。

所有受指令影响的标志的符号下方都将有一个星号(*)出现。标志是如何受到影响的，取决于指令所做的事情。你必须通过注释部分进行推测。当某条指令没有影响任何标志时，在标志迷你图中将出现单词<none>。

在这里的示例页面中，迷你图表明 NEG 指令影响溢出标志、符号标志、零标志、辅助进位标志、奇偶标志和进位标志。这些标志如何受到影响，取决于求补操作对于特定操作数的执行结果。这些可能性被总结在第二段的注释中。

7.6　NEG：求补(求补码；即，与-1 相乘)

影响的标志

```
O D I T S Z A P C   OF：溢出标志    TF：陷阱标志    AF：辅助进位标志
```

```
F  F  F  F  F  F  F  F  F   DF: 方向标志    SF: 符号标志    PF: 奇偶标志
*           *  *  *  *  *   IF: 中断标志    ZF: 零标志      CF: 进位标志
```

合法形式

```
NEG r8
NEG m8
NEG r16
NEG m16
NEG r32 386+
NEG m32 386+
```

示例

```
NEG AL
NEG DX
NEG ECX
NEG BYTE [BX]      ; 将位于[BX]处的长度为一个字节的量求补
NEG WORD [DI]      ; 将位于[BX]处的长度为一个字的量求补
NEG DWORD [EAX]    ; 将位于[EAX]处的长度为一个双字的量求补
```

注解

这是将一个值与-1 相乘的汇编语言等值。请记住，求补并不是简单地将操作数中的每个位取反。(另一个指令，NOT，是这样做的。) 这个过程也被称为生成一个值的补码。一个值的补码与该值相加，产生的结果为零。-1 = $FF; -2 = $FE; -3 = $FD，等等。

如果操作数为 0，则 CF 是被清零，ZF 被设置，否则，CF 被设置，ZF 被清零。如果操作数包含最大负值(8 位最大负值为-128；16 位最大负值为-32 768)，则操作数不变，但是 OF 和 CF 被设置。如果结果为负，SF 被设置，否则 SF 被清除。如果结果的低 8 位包含偶数个设置为 1 的位，PF 的值被设置，否则，PF 被清零。

注意，对于内存中的数据，必须使用大小说明符(BYTE, WORD, DWORD)！

```
r8 = AL AH BL BH CL CH DL DH     r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS           r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                 m16 = 16 位内存数据
m32 = 32 位内存数据               i8 = 8 位立即数
i16 = 16 位立即数                 i32 = 32 位立即数
d8 = 8 位有符号位移               d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

7.6.1 合法形式

每个给定的助记符都代表一个单个的 x86 指令，而每个指令都可能包括一种以上的合法形式。一个指令的形式会根据传递给它的操作数的类型和顺序而发生变化。

实际上不同的形式代表不同的二进制操作码。例如，在最底层，POP AX 指令是二进制数 058h，而 POP SI 指令是二进制数 05Eh。多数操作码不是单个的 8 位值，大多数至少有两个字节长，而且往往是四个或更多字节。

有时也会有指令及其操作数的特殊情况并且操作数比一般情况短。例如，用于交换两个操作数的内容的 XCHG 指令，当它的一个操作数存放在寄存器 AX 中时，就有一种特殊情况。任何将 AX 作为一个操作数的 XCHG 指令都可用一个简单的字节操作码代表。XCHG 的一般形式(例如，XCHG r16，r16)中，总是两个字节长。这意味着实际上有两种不同的操作码，能够完成对于一个给定的操作数组合的操作(例如，XCHG AX,DX)。没错，一些汇编编译器拥有足够的智慧，它们能够在所有特定情况下选择尽可能短的形式。如果你正在手工编译一个原始的操作码字节序列，例如，在更高级别的语言中使用的内联汇编语句，你需要注意一些特殊情况，在“合法形式”小节中对所有的特殊情况都进行了说明。

当你想使用具有一组特定操作数的指令时，务必检查该指令的参考指南中的“合法形式”小节，以确保该组合是合法的。与以前的糟糕的 DOS 时代相比，现在有更多的合法形式，很多保留下来的限制都涉及段寄存器，当编写一般的 32 位保护模式用户应用程序时，你将无法使用它。例如，MOV 指令不能将数据从某个内存位置移动到另一个内存位置，并且在实模式下存在一些关于如何在段寄存器中存放数据的限制。

在 NEG 指令的示例参考页中可以看到，段寄存器不能作为 NEG 指令的操作数。(如果可以，就应该有一个 NEG sr(将在下一节讨论的)项出现在合法形式列表中。)

7.6.2 操作数符号

在“合法形式”小节中，用于表明操作数本质的符号都在附录 A 中每一条指令页的底部进行了总结。它们接近不言自明，但我还是想在这里花点时间将其稍微展开：

- **r8：**一个 8 位的半寄存器，AH、AL、BH、 BL、 CH、CL、DH 或者 DL 中的一个。
- **r16：**一个 16 位通用寄存器，AX、BX、CX、DX、 BP、 SP、 SI 或者 DI 中的一个。
- **r32：**一个 32 位通用寄存器，EAX、 EBX、ECX、EDX、EBP、ESP、ESI 或者 EDI 中的一个。
- **sr：**一个段寄存器，CS、DS、SS、ES、FS 或者 GS。
- **m8：**一个 8 位的字节内存数据。
- **m16：**一个 16 的位字内存数据。
- **m32：**一个 32 的位字内存数据。
- **i8：**一个 8 位的字节立即数。
- **i16：**一个 16 位的字立即数。
- **i32：**一个 32 位的字立即数。
- **d8：**一个 8 位的有符号位移。我们现在还没有涵盖这些内容，但是位移是一个代码的当前位置与该代码中的另一个你想跳转到的位置之间的距离。它是有符号值(即，或者是一个正数，或者是一个负数)，因为一个正位移跳转到内存中的较高(向前)位置，而负位移则跳转到内存中的较低(向后)位置。我们将在后面详细研究这个概念。
- **d16：**一个 16 位的有符号位移。再次的，用于 jump 指令和 call 指令。
- **d32：**一个 32 位有符号位移。

7.6.3 示例

“合法形式”小节主要展示什么样的操作数组合对于给定的指令而言是合法的，示例部分则展示一些实际使用中的指令示例，就像它们在汇编语言程序中的编码一模一样。我试图为每一条指令提供一个良好的范例取样，尽量体现该条指令不同可能性的范畴。

7.6.4 注解

参考页的“注释”小节简单描述了指令的活动，并提供了一些信息，包括它如何影响标志、如何可能被限制使用，以及所有需要记住的其他细节，特别是那些初学者容易忽视或误解的内容。

7.6.5 这里没有包含的内容

附录 A 不同于大多数详尽的汇编语言参考资料，它不包括操作码的二进制编码信息，也没有表明该指令的每种形式使用多少机器周期的相关内容。

一条指令的二进制编码是 CPU 消化和识别的机器指令的实际二进制字节序列。我们称为 POP AX 的指令被机器视为二进制数 58h。我们称之为 ADD SI,07733h 的指令被机器视为 4 个字节的序列：81h 0C6h 33h 77h。机器指令可以被编码到任意地方，根据它们是什么指令以及操作数是什么，从一个到四个(有时更多) 二进制字节不等。展示用于决定任一给定指令的编码的系统非常复杂，它的成员字节必须被一位一位地设置，而且这些位来自几张大表。所以我决定，本书不是讨论这一特定内容的地点，因此，在附录参考资料中不考虑编码信息。(正是这一点才让 Intel 指令参考书变得像现在那么庞大。)

最后，我没有在本书的任何地方涵盖指示任意给定指令需要消耗多少机器周期的内容。机器周期是其中一个主时钟的脉冲，是它让个人电脑能够执行“个人魔术”。每个指令都使用一定数量的这种周期来完成自己的工作，根据一些我不会在本书中解释的标准，这个数字有所不同。更糟的是，给定指令所使用的机器周期数从一个 Intel 处理器型号到另一个型号也有所不同。在奔腾机器上，一条指令使用的机器周期可能比 486 上使用的少几个，或者少更多。(一般来讲，这些年来 x86 指令已经进化为使用更少的时钟周期，但是事实上并非对于每一条指令都是如此。)

此外，像 Michael Abrash 在他的巨著《Michael Abrash 的图形编程黑皮书》中所说的一样，要想计算一个给定系列的指令将花费多长时间来执行，仅仅知道单个指令的执行周期要求还远远不够，即使是专家级别的汇编语言编程员。CPU 缓存、预取指、分支预测、超线程和任意数量的其他因素结合在一起，相互作用，使得进行这样的计算几乎不可能，除非从广义上来讲。他和我都认为，这不是一个适合初学者的问题，如果想更多地了解与之相关的内容，我建议你找到他的书，然后自己去研读。

第 8 章

我们的崇高目标

创建能够工作的程序

他们不是无端地称它为“汇编”的。面对编写一项汇编语言程序的任务，我想起了圣诞节早晨的场景：你已经把 1567 个小型金属零件从一个标识为“陆地鲨鱼超级自行车”(需要装配)的大箱子中倒了出来，接下来，你必须以某种方式把所有这些零件组装在一起，一个也不剩。(在此期间，孩子们得到的乐趣似乎比在箱子里玩耍还要多。)

我其实已经差不多把创建一个汇编语言程序所需要的所有内容都讲解完毕了。不过，仍然有一个从这里到那里的不平凡的飞跃，你面临的是许多边缘锋利的小部件，而这些部件的组合方式可以有无穷多种，其中大多数是错误的，有一些是可行的，但是只有少数是理想的选择。

因此，这就是我们的计划：在本章中，我将向你介绍一个完整的，可操作的“陆地鲨鱼超级自行车”——然后我会在你的眼前把它拆散。这是学习汇编的最佳方法：拆解那些知道工作原理的专家所编写的程序。在本章中，我们将拆开几个程序，希望当本章结束的时候，你能够凭借自己的力量重新再将它们组装起来。

8.1 汇编语言程序的基本框架

回到第 5 章的程序清单 5-1，我呈现的也许是最简单的能够正确运行在 Linux 下的程序，它能够做某些看得见的事情，并且仍然具有很好的可理解性和可扩展性。从那时起，我们一直通过 Insight 调试器观察沙盒中的指令。这是一个熟悉单个指令的好办法，但是很快沙盒就不够用了。既然你已经掌握了最常见的 x86 指令(并且知道如何建立一个沙盒来进行试验，进而逐渐解其他指令)，我们需要

继续前往更为完整的程序。

就像你在运行 eatsyscall 程序时看到的一样，它会在显示屏上显示一(短)行文字：

```
Eat at Joe's!
```

而为了实现这一点，你不得不向汇编编译器提供 35 行文字！这 35 行中有许多是注释，严格意义上来讲，这些注释不是必要的，但是它们作为内部文档，能够让你在六个月后或者一年后仍然了解这个程序在做什么(或者，更重要的是，它如何完成所需完成的任务)。

下面介绍的程序与清单 5-1 中的程序相同，但是我在这里仍然要重复它，这样你就不必在接下来的讨论中来回翻书了：

```
; 可执行程序名:      EATSYSCALL
; 版本: 1.0
; 创建日期:          1/7/2009
; 最后修改日期:      1/7/2009
; 作者:              Jeff Duntemann
; 描述:              Linux 下的一个简单的汇编应用程序，使用 NASM 2.05,
;                    演示了如何使用 Linux INT 80H 系统调用来显示文本。
;
; 使用这些命令生成:
; nasm -f elf -g -F stabs eatsyscall.asm
; ld -o eatsyscall eatsyscall.o
;

SECTION .data               ; 包含已初始化数据的部分

EatMsg: db "Eat at Joe's!",10
EatLen: equ $-EatMsg

SECTION .bss                ; 包含未初始化数据的部分

SECTION .text               ; 包含代码的部分

global _start               ; 连接程序需要根据它找到入口点!

_start:
      Nop                   ; 无操作指令使 gdb 非常高兴 (见正文)
      mov eax,4             ; 指定 sys_write 系统调用
      mov ebx,1             ; 指定文件描述 1: 标准输出
```

```
mov ecx,EatMsg       ; 传递显示信息的偏移地址
mov edx,EatLen       ; 传递显示信息的长度
int 80H              ; 进行系统调用以将文本输出到标准输出

mov eax,1            ; 指定 Exit 系统调用
mov ebx,0            ; 返回一个零编码
int 80H              ; 进行系统调用来终止程序
```

8.1.1　最开始处的注释块

汇编语言编码的目的之一，是使用尽可能少的指令来完成某项工作。这并不意味着创建尽可能短的源代码文件。源文件的大小与将其编译后得到的可执行文件的大小毫无关系！放入文件的注释越多，当下次再捡起它的时候，你就越能更快地想起它的内部工作原理。我想你会发现，一个复杂的汇编语言程序的逻辑在你的脑海里的冷却速度快得惊人。在工作于其他项目不足 48 小时之后，我已经回到原来的汇编项目，但是却必须经过一番努力之后才能重新回到全速开发状态。

注释既不浪费时间，也不浪费空间。IBM 曾经建议“每行代码一行注释”。这很好，并应被视为汇编语言工作最起码的组成内容。一个更好的(我在后面较为复杂的例子中所遵循的)习惯是在每一行代码的右侧使用一短行注释，同时在共同完成某一单独任务的每个指令序列的开始位置添加注释块。

每一个程序的顶部应该是一个标准化的注释块，其中包含一些重要的信息：

- 源代码文件的名称
- 可执行文件的名称
- 创建该文件的日期
- 最后一次修改该文件的日期
- 程序编写者的名字
- 用于创建它的汇编编译器的名称和版本
- 程序或者库文件所做事情的概要说明。需要多少空间你就用多少。它并不会影响可执行程序的大小和速度
- 如果使用生成文件(你应该那么做。)的话，还应该有一个来自于生成文件的、用于生成该文件的命令的副本

对于程序开头的注释块而言，挑战在于更新它，以使其反映项目的当前状态。没有工具能够自动这样做。这项任务由你来负责。

8.1.2 .data 段

用 NASM 编写的运行在 Linux 下的普通用户空间程序分为三个段。这些段在程序中的排列顺序并不重要，但是根据习惯，.data 段在最上面，其次是.bss 段，最后是.text 段。

.data 段包含了已初始化的数据项的数据定义，已初始化数据在程序开始运行前就拥有自己的值，这些值是可执行文件的一部分，当可执行文件被加载到内存中用于执行时，它们被加载到内存中。你不必非要将它们与其值一起加载，除了将程序作为一个整体而加载到内存中所需的机器周期之外，并不需要额外的时间来创建它们。

关于.data 段，最重要的是要记住，定义的初始化数据项越多，可执行文件就会越大，运行它的时候，也就需要更长的时间才能将它从磁盘里加载到内存中。

稍后将马上详细研究如何始定义初化数据项。

8.1.3 .bss 段

并不是所有数据项在程序开始运行之前都拥有值。例如，从磁盘文件中读取数据时，当数据从磁盘中出来之后，需要有一个存放的地方。那样的数据缓冲区是在程序的.bss 段中定义的。你为缓冲区预留一定数量的字节，然后给缓冲区取一个名字，但是并不指明缓冲区中将要出现什么值。

.data 段中定义的数据项和.bss 段中定义的数据项之间有一个重要的差别：.data 段中定义的数据项将会添加到可执行文件的大小上，而.bss 段中定义的数据项则不会。我们可以在.bss 段中定义一个 16 000(或更多，有时会比它多得多)字节的缓冲区，而可执行文件的大小几乎不变(除了大约 50 个字节用于描述以外)。

Linux 加载程序把程序带入内存的方式使这成为可能。当生成可执行文件时，Linux 连接器向描述所有已定义符号的文件添加一些信息，包括用来命名数据项的符号。加载程序知道哪些数据项没有初始值，当把可执行文件从磁盘加载到内存时，它为这些数据项分配空间，而具有初始值的数据项则与其初始值一起读入。

最简单的程序 eatsyscall.asm 并不需要任何缓冲区或者其他未初始化的数据项，并且从技术上来讲，并不需要定义一个.bss 段。拥有一个空的.bss 段并不会

使可执行文件变得更大，删除一个空的.bss 段也不会使可执行文件变得更小。

8.1.4 .text 段

真正组成程序的机器指令存放在.text 段中。一般情况下，在.text 段中不进行数据项定义。.text 段包含名为标号(label)的符号，这些符号用于标识跳转和调用的程序代码位置，但是位于指令助记符之外。

所有的全局标号必须在.text 段中声明，否则该标号就不能被 Linux 连接程序和 Linux 加载程序在你的程序之外看到。让我们更详细地了解一下标号问题。

8.1.5 标号

标号是一种类型的书签，用于描述程序代码中的某个位置，由于它拥有一个名字，因而比毫无装饰的内存位置更加容易记忆。标号用于标识转移指令应该跳转到的位置，也可为可调用的汇编过程起个名字。我将在后面的章节中讲解相关的实现原理。

这里是我们需要了解的关于标号的最重要内容：

- *标号必须以字母开头；否则用下划线、句号或者问号开头。*后三种对于汇编编译器来说，有着特殊的含义，所以，在了解 NASM 如何解释它们之前，不要使用它们。
- *当定义标号时，后面必须跟一个冒号。*这主要是告诉 NASM，被定义的标识符是一个标号。如果没有冒号的话，NASM 会像凌空踢球一样一脚把它踢出来，不会标记任何错误，但是如果有冒号定在那里的话，就可以防止一个敲入有误的指令助记符被误当作标号。所以请使用冒号！
- *标号是大小写敏感的。*所以，yikes:、Yikes:、和 YIKES: 是三个完全不同的标号。这一点与很多其他语言(尤其是 Pascal)中的经验不同，所以一定要记住。

……在后面部分，你将看到这样的标号被用做转移指令和 call 指令的目标操作数。例如，以下机器指令将把指令执行流程转移到标号 GoHome:所标记的位置：

```
jmp GoHome
```

注意，这里没有使用冒号。冒号的唯一出现位置是在定义标号的地方，而不

是调用标号的地方。可以这样认为：当你要标识一个位置时，请使用冒号，当你将要去往一个位置时，不要使用冒号。

在 eatsyscall.asm 中只有一个标号，而且这个标号有些特殊。_start 标号标识程序从何处开始。每一个 Linux 汇编语言程序都必须以这种方式进行标识，并且必须精确地采用_start 标号。(它是大小写敏感的，所以不要尝试使用_START 或者_Start。)此外，这个标号必须在.text 段的顶部标记为全局的，如前所示。

这是 Linux 操作系统的要求。Linux 下的每一个可执行程序都必须在某处有一个_start 标号，不管编写它的语言是 C、Pascal 还是汇编。如果 Linux 加载程序找不到这个标号，它就不能正确地加载该程序。Global 限定符告诉连接器让_start 标号在程序的边界之外可见。

8.1.6　已初始化变量

.data 段中的标号 EatMsg 定义了一个变量。确切地说，EatMsg 是一个字符串变量(后面内容中将不断用到这种类型)，但是和所有变量一样，它是一种已初始化数据项：已经带有一个值，而不仅仅是一个用于将来某一刻存放数据的内存空间。变量通过数据定义指令与标识符相关联。数据定义指令使用形式如下：

```
MyByte db 07h                 ; 8 位大小
MyWord dw 0FFFFh              ; 16 位大小
MyDouble dd 0B8000000h        ; 32 位大小
```

可以将 DB 指令理解为“定义字节”。DB 预留一个字节的内存空间用于数据存储。可以将 DW 指令理解为“定义字”。DW 预留一个字(16 位或 2 个字节)的内存空间用于数据存储。可以将 DD 指令理解为“定义双字”。 DD 在内存中预留一个双字的空间用于数据储存，通常用于完整的 32 位内存地址。

8.1.7　字符串变量

字符串变量是一种有趣的特例。所谓字符串，是指位于连续内存区域中的一个字符序列，或者一串字符。在 eatsyscall.asm 中定义了这样一个字符串变量：

```
EatMsg: db "Eat at Joe's!”,10
```

数据定义指令通常会预留一定数量的内存，但是字符串是这一原则的例外。DB 指令通常只预留一个字节，而字符串却可以为任意长度。因为不存在预留 17

个，或 42 个字节的数据指令，所以字符串只需通过在起始处关联一个标号来进行定义。EatMsg 标号和 DB 指令在内存中指定一个字节作为字符串的起点。而字符串中的字符数则告诉汇编编译器为该字符串预留多少个字节的存储空间。

单引号(')字符和双引号(")字符都可用于描述字符串，具体使用哪个随你选择，除非你定义的字符串本身就包含一个或多个引号字符。注意，在 eatsyscall.asm 程序中，字符串变量 EatMsg 包含一个单引号字符，作为一个单引号来使用。因为该字符串包含一个单引号字符，所以必须用双引号来叙述它。反过来也一样：如果定义一个包含一个或多个双引号字符的字符串，必须用单引号字符来描述它：

```
Yukkh: db 'He said, "How disgusting! " and threw up.',10
```

通过用逗号分隔各个字符串，可以将几个单独的子串合并成一个字符串变量。下面这种定义字符串变量的方式是完全合法的(而且有时非常有用)：

```
    TwoLineMsg: db "Eat at Joe's...” ,10,"...Ten million flies can't
ALL be wrong!” ,10
```

在上述示例中的字符串里，立即数 10 是做什么用的呢？在 Linux 文本工作中，行结束(EOL)字符的数字值为 10，它告知操作系统，提交给 Linux 控制台用于显示的行在哪里结束。后续文本将在下一行开始处进行显示。在变量 TwoLineMsg 中，两个子字符串之间的 EOL 字符将引导 Linux 在控制台的某一行显示一个子串，并在该行下方另起一行显示第二个子串：

```
Eat at Joe's!
Ten million flies can't ALL be wrong!
```

我们可以将这样的数字串联到一个字符串内部，但是必须记住，如同 EOF 一样，它们并不会被显示为数字。字符串是一个由字符构成的串。追加到字符串的数字将被大多数操作系统例程解释为一个 ASCII 字符。数字和 ASCII 字符之间的对应关系已经在附录 B 中给出。要想在一个字符串中显示数字，必须将其表示为 ASCII 字符，要么用字符型立即数(如“7”)，要么用与之等值的 ASCII 字符(如 37h)。

在一般的汇编工作中，几乎所有的字符串变量都可通过使用 DB 指令进行定义，并且视为字节字符串。(一个 ASCII 字符的大小是一个字节。)也可以使用 DW 或 DD 来定义字符串变量，但是对它们的处理与使用 DB 定义的字符串有些

不同。考虑这些变量：

```
WordString: dw 'CQ'
DoubleString: dd 'Stop'
```

DW 指令定义了一个长度为一个字的变量，一个字(16 位)可以存放 2 个 8 位字符。类似地，DD 指令定义了一个长度为双字(32 位)的变量，它可能存放 4 个 8 位字符。当加载这些已命名字符串变量到寄存器时，它们将被以不同的方式进行处理。考虑下面两条指令：

```
mov ax,wordstring
mov edx,DoubleString
```

在第一条 MOV 指令中，字符“ CQ”被放到寄存器 AX 中，其中，“C”放在 AL 中，“Q”放在 AH 中。在第二条 MOV 指令中，4 个字符“Stop”被按照小端字节序加载到 EDX 中， 其中，“S ”在 EDX 的最低序字节，“ t”在第二低字节中，以此类推。此类情况比用 DB 定义字符串少见些(用处也少一些)，你会发现自己很少使用它。

由于 eatsyscall.asm 中没有包含任何未初始化的数据，我将会推后对于此类定义的讨论，直到我们看完下面的示例程序。

8.1.8　通过 EQU 和$推导字符串的长度

在 eatsyscall.asm 文件中，如下所示的 EatMsg 定义是一个有趣的构造：

```
EatLen: equ $-EatMsg
```

这是一个编译时计算的例子。我们在这里所做的事情是：计算字符串变量 EatMsg 的长度，并且使该长度值可以通过标号 EatLen 来进行访问。不管在程序的任何地方，只要用到 EatMsg 的长度，都可以使用标号 EatLen。

包含 EQU 指令的语句叫作等值语句。等值语句是一种将一个值与标号相关联的方法。然后这样的标号就可以被像 Pascal 中的已命名常量一样处理了。每当汇编编译器在编译过程中遇到一个等值语句，它都将该等值语句的值与其名字进行交换。例如：

```
FieldWidth equ 10
```

前面的语句告诉汇编编译器标号 FieldWidth 代表数值 10。一旦该等值已经

被定义，下面的两个机器指令就完全相同：

```
mov eax,10
mov eax,FieldWidth
```

这样做有两个优点：

- 等值通过为一个值使用描述性的名字而使指令更容易理解。我们知道 10 这个值在此处的含义：它是字段的宽度。
- 等值使程序更容易修改。如果在某种情况下，字段宽度从 10 变化为 12，我们只需修改源文件中的一行代码，而不是每一处访问该字段宽度的地方。

不要低估了第二个优点的价值。一旦程序变得越来越庞大，越来越复杂，你可能会发现自己在一个程序中使用某一特定值可达数十次或者数百次。你可以为该值编写一条等值语句，然后只需修改一行代码即可修改这个被用了 267 次的值；或者通读代码，然后一个一个地修改所有这 267 处使用该值的地方——除了五六个漏掉的之外，而这将在下次编译和运行程序时导致大的混乱。

将汇编语言计算与等值语句相结合使得我们可以采用非常简单的方式完成一些奇妙的事情。正如稍后我将会讲解的一样，要想在 Linux 下显示一个字符串，我们不仅需要向操作系统传递字符串的地址，而且需要传递它的长度。可通过以下方式创建一个字符串长度的等值：

```
EatMsg db "Eat at Joe's!",10
EatLen equ 14
```

这样做是可行的，因为 EatMsg 字符串其实是 14 个字符，包括 EOL 字符。但是假设 Joe 将他的速食店卖给了 Ralph，你需要将 Joe 换成 Ralph。你不仅需要修改广告信息，而且需要修改它的长度：

```
EatMsg db "Eat at Ralph's!",10
EatLen equ 16
```

忘记更新 EatLen 等值使之与新的消息长度相等的概率有多少呢？如果你经常做这类事情，肯定会有这样的情况发生。对于编译时计算而言，你只需修改字符串变量的定义，而它的长度则由 NASM 在编译时自动计算。

如何实现呢？采用下面这方法即可：

```
EatLen: equ $-EatMsg
```

它的实现全靠那个神奇的、通过小小的$符号来表示的“这里”令牌。正如前面讲解的一样，在汇编时，NASM“咀嚼”源代码文件，然后生成一个以.O 为扩展名的中间文件。$令牌标识 NASM 在中间文件中的位置(而不是在源代码文件中的位置！)。标号 EatMsg 标志着广告语字符串的开始位置。紧跟在 EatMsg 的最后一个字符后面的是标号 EatLen。记住，标号不是数据，而是位置，并且在汇编语言中，它是一个地址。当 NASM 到达标号 EatLen 时，$的值代表 EatMsg 的最后一个字符后面的那个位置。编译时计算将会采用$令牌(当计算完成后，它将包含刚刚超过 EatMsg 字符串结尾的那个位置)所代表的位置，并且将其与 EatMsg 字符串的开始位置相减。结束处 - 开始处= 长度。

每次编译文件时都执行这种运算，因此任何时候 EatMsg 的内容发生改变，EatLen 的值都将被自动重新计算。你可以随心所欲地修改字符串中的文字，根本不用为修改任何出现在程序中的长度值而担心。

编译时计算还有一些其他的用途，但是这是最常见的一种，而且是一个当你作为初学者时很可能使用的用途。

8.2　通过堆栈实现后进先出

eatsyscall.asm 这个小程序做的事情并不多：它在 Linux 控制台上显示一个简短的文本字符串。但是，讲解它如何完成这样一件简单的事情，却需要花些心思，甚至在开始讲解之前，我必须介绍一个不仅是 x86 架构中的、其实也是所有计算机计算中的关键概念：堆栈。

堆栈是一个直接构建在 x86 硬件之上的存储机制。英特尔没有发明它；自从 20 世纪 50 年代以来，堆栈一直是计算机硬件的组成部分。它的名称很恰当，为了引入一个恰当的比喻，我可以追溯到高中时代，那时候，我在位于芝加哥西北侧的复活医院里做洗碗工。

8.2.1　每小时 500 个盘子

当时，在医院的洗碗室中有许多种不同的工作，但是我大部分时间所做的事情是把干净的盘子从一台 180 洗碗机里出来的、不断喷着蒸汽的传送带上移走。这是高温作业，但是，它的讨厌程度比起将脏兮兮的盘子塞进机器的另一端来要小多了。

当你需要每小时从洗碗机中拉出 500 个盘子时，最好找个地方高效地存放它

们。很明显，你可以简单地将它们堆放在桌子上，但是，将陶瓷做的盘子堆放在任何毛手毛脚的十几岁男孩所习惯的地方都将导致餐具的混乱。医院取而代之的是使用一批带有小轮的不锈钢橱柜，每个橱柜配备一个或者多个能够从顶部访问的、装了弹簧的圆形活塞。当你取出一摞盘子时，可将它们向下推到活塞里。活塞的弹簧被调整成这样：后来添加的盘子的重量可将整堆盘子压下去，刚好腾出足够的位置，以使橱柜顶部能够不断放入新的盘子。

每个活塞大约能够盛放 50 个盘子。我们把活塞升起到紧挨洗碗机的地方，待装满盘子之后，再将其送回到厨房中，在那里，工作人员将使用这些干净的盘子为病人配制下一餐的饮食。

跟踪某一轮中第一个从刷碗机中出来的盘子的路径具有一定的启发性。这个盘子第一个进入活塞，随后被橱柜存放的其余 49 个盘子往下推到活塞的底部。当橱柜被送进厨房后，厨房的工作人员在配置用餐托盘时，他们将盘子从橱柜中一个接一个地拿出来。第一个从橱柜中出来的是最后一个放进去的盘子；最后一个出来的盘子是第一个放进去的盘子。

x86 堆栈(和在其他计算机硬件体系结构中的大部分其他堆栈)中的情况和上述情形非常相像。它被叫作后进先出或者 *LIFO* 栈。取代盘子的是，我们在堆栈顶部压入很多数据块，并且这些数据块将一直待在堆栈中，直到我们按照相反的顺序将它们弹出。

堆栈并不存于某些单独的 CPU“壁橱”中。它就存在于普通的内存之中，事实上我们称之为“堆栈”的东西实际上是一种内存数据的管理方式。堆栈是一个地方，我们可以暂时放入其中一个或两个(或许多个)32 位双字，并在稍晚的时候再取出它们。它的主要好处是不需我们给出存储数据的名称。我们将数据放入堆栈，后面不是通过内存地址而是通过位置来找回它们。

涉及堆栈使用的术语反映了我的洗碗机比喻：当把数据放入堆栈时，我们说，我们把数据压入堆栈；当从堆栈中取出数据时，我们说，我们把数据弹出堆栈。堆栈随着数据被压入或者弹出而增长或者缩小。最新压入堆栈的项被认为是在“堆栈的顶部”。当从堆栈弹出一个项时，我们得到的是位于堆栈顶部的那一项。我已经从概念的层面将其绘制在图 8-1 中。

在 x86 体系结构中，堆栈的顶部由名为堆栈指针寄存器的寄存器来进行标识，该寄存器的正式名称是 ESP。它是一个 32 位寄存器，里面存放着最后一个压入栈中的项的内存地址。

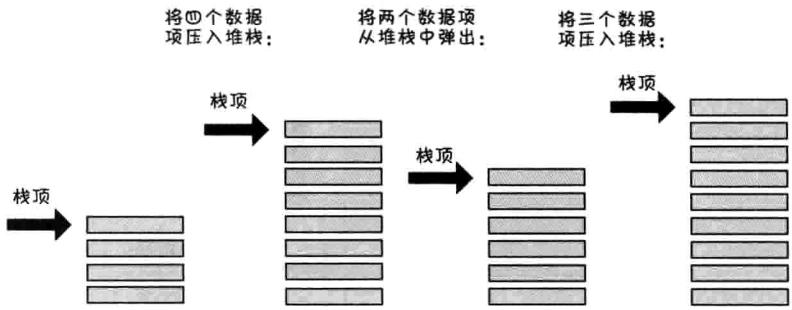


图 8-1　堆栈

8.2.2　堆栈的内容上下颠倒

x86 堆栈基本上是上下颠倒的，这使得堆栈的可视化效果变得有点复杂。如果画一幅最低地址在底部，最高地址在顶部的内存区域图，那么堆栈是从天花板开始的，随着数据项被压入到堆栈中，堆栈朝着低内存的方向向下增长。

图 8-2 从广义上显示了当程序运行时，Linux 如何组织分配给程序的内存。在内存底部的是程序中定义的三个段：.text 段位于最低的地址处，其次是.data 段，再次是.bss 段。堆栈一直位于程序的内存块的另一端。位于.bss 段的末尾和堆栈的顶部之间的内存基本上是空的。

C 程序经常使用这种剩余内存空间来为那些位于堆内存中的、“已在运行中”的变量分配空间。汇编程序也可以那么做，尽管并不像听起来那么容易，我也不打算在本书涵盖相关内容。需要记住的最重要的事情是：堆栈和程序(代码和已命名数据)在沙盒的相对角落里循规蹈矩地“玩耍”。堆栈朝着程序的剩余部分增长，但是除非你正在做一些真正惊人的事情，或者是愚蠢的事情，堆栈很少或者根本没有机会增大到能够撞到程序的已命名数据项或者机器指令的程度。如果出现这种情况，Linux 将会冷静地发出分段错误并且终止程序。

对于图 8-2，我的唯一警告是：程序段与堆栈的相对大小不应按照字面理解。程序代码可能有上千个字节，一个中等汇编程序中的数据可达成千上万个字节，但是所用的堆栈仍然相当小：最多几百个字节，一般低于这个值。

注意，当程序开始运行时，堆栈并不完全是空的。一些有用的东西在那里等着你，我将在稍后进行讲解。

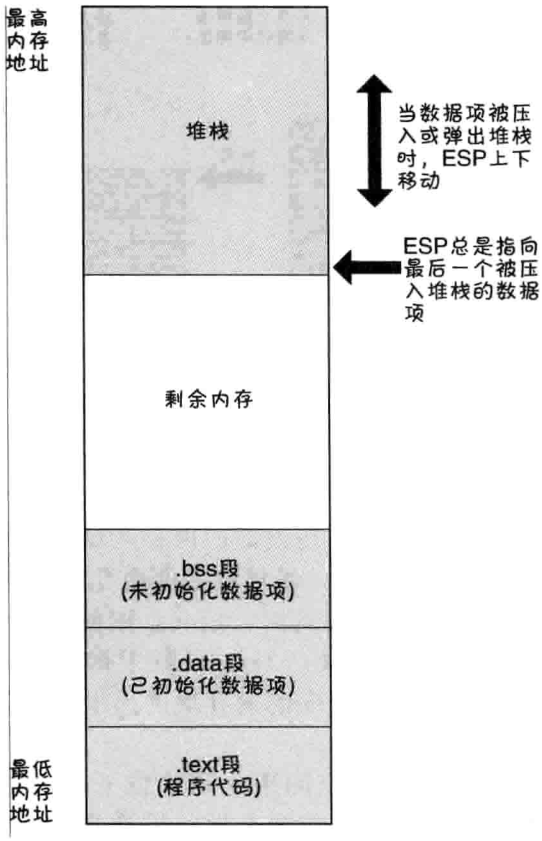


图 8-2 程序内存中的堆栈

8.2.3 Push-y 指令

我们可以采用几种方法将数据放入堆栈中，但最直接的方法涉及一组相关的机器指令，一共有 5 个：PUSH，PUSHF，PUSHFD，PUSHA 和 PUSHAD。所有指令都以相似的方式工作，它们之间的主要不同在于压入堆栈的数据：

- PUSH 将一个在源代码中指定的 16 位或者 32 位寄存器值或者内存值压入堆栈。
- PUSHF 将 16 位标志寄存器的值压入堆栈。
- PUSHFD 将 Eflags 寄存器的全部 32 位值压入堆栈。
- PUSHA 将所有的 8 个 16 位通用寄存器压入堆栈。

- PUSHAD 将所有的 8 个 32 位通用寄存器压入堆栈。

下面是 PUSH 系列指令的一些使用举例：

```
pushf       ; 将标志寄存器压入堆栈
pusha       ; 将 AX、 CX、DX、BX、 SP、 BP、 SI 和 DI 按此顺序一次性压入堆栈
pushad      ; 将 EAX、 ECX、 EDX、 EBX、 ESP、 ESP、 EBP、 ESI 和 EDI
              一次性压入堆栈
push ax     ; 将 AX 寄存器压入堆栈
push eax    ; 将 EAX 寄存器压入堆栈
push [bx]   ; 将存放在 BX 里的内存中的字压入堆栈
push [edx]  ; 将存放在 EDX 里的内存中的双字压入堆栈
push edi    ; 将 EDI 寄存器压入堆栈
```

请注意，PUSHF 和 PUSHFD 不使用任何操作数。如果尝试给它们传递操作数，将产生一个编译错误；这两条指令将标志寄存器的值压入堆栈，而且这就是它们能做的所有事情。

PUSH 采用以下方式处理 32 位操作数：首先，ESP 递减 32 位(4 个字节)，指向堆栈段的一个四字节长的空白区域。然后，将即将压栈的数据写入 ESP 指定的内存中。瞧！数据在堆栈上很安全，ESP 已经朝着内存底部的方向爬行了 4 个字节。PUSH 也可以将 16 位值压入堆栈，当它做这件事情的时候，唯一区别是 ESP 移动的是 2 个字节而不是 4 个字节。

PUSHF 以同样的方式工作，但是它压入的是 16 位的标志寄存器。

PUSHA 也以同样的方式工作，但是它将所有的 8 个 16 位通用寄存器一次性压入堆栈，因而一下使用 16 个字节的堆栈空间。PUSHA 从 286 处理器开始才添加到指令集，在 8086/8088 处理器中并不存在。

PUSHFD 和 PUSHAD 从 386 处理器开始才添加到 x86 指令集。除了压入的是 32 位寄存器的值，而不是 16 位寄存器的值之外，它们与它们的 16 位替代者以同样的方式工作。PUSHFD 将 32 位的 Eflags 寄存器的值压入堆栈。PUSHAD 一下子将所有的 8 个 32 位通用寄存器的值都压入堆栈。

因为 Linux 至少需要运行在 386 上，所以可以假设，任何 Linux 安装都支持 PUSHA、PUSHFD 和 PUSHAD。

SP 初始位置与当前位置(栈顶)之间的所有内存包含真正的数据，这些数据被显式地压入堆栈，并且以后可从堆栈中弹出。其中有些是在运行操作系统之前被操作系统压入堆栈的，我们将在本书稍后部分讨论这一点。

哪些数据可以压入堆栈，哪些数据不可以压入堆栈，这比较复杂，取决于你

所使用的 CPU。任何 16 位和 32 位通用寄存器都可以被单独压入堆栈。但是 x86 CPU 不能将 8 位寄存器压入堆栈。换句话说，你不能将 AL 或 BH 或者任何其他 8 位寄存器压入堆栈。立即数可以被压入堆栈，但是只有在 286 或者更高的处理器上才行。(在 Linux 操作系统下这将永远为真。)用户模式 Linux 程序任何情况下都不能将段寄存器压入堆栈。

跟踪所有这一切曾经是 DOS 时代的一个难题，但是，目前你不太可能在比 386 还早的处理器上运行代码，也绝不可能在 Linux 下运行代码。

8.2.4　POP 指令

一般来说，压入堆栈的内容必须要弹出来，否则你可能以若干种不同的麻烦之一来结束程序。将堆栈中的数据项弹出可通过另外一组指令五重奏来完成：POP、POPF、POPFD、POPA 和 POPAD。正如预想的那样，POP 是一个通用的、一次处理一个数据的弹出指令，而 POPF 和 POPFD 专门用于从堆栈中弹出标志寄存器。POPA 从堆栈中弹出 16 个字节的数据到 8 个 16 位的通用寄存器中。POPAD 是 PUSHAD 的反向操作，它将栈顶的 32 个字节弹出到 8 个 32 位通用寄存器中。以下是一些例子：

```
popf        ; 从栈顶弹出两个字节到标志寄存器中。
popa        ; 从栈顶弹出 16 个字节到 AX、CX、DX、 BX、BP、SI 和 DI...
            ; 但是 SP 中的值不变!
popad       ; 从栈顶弹出 32 个字节到 EAX、ECX、EDX、EBX、 EBP、ESI
            ; 和 EDI...但是 ESP 中的值不变!!!
pop cx      ; 从栈顶弹出 2 个字节到 CX 中。
pop esi     ; 从栈顶弹出 4 个字节到 ESI 中
pop [ebx]   ; 从栈顶弹出 4 个字节到 EBX 中的值所指定的内存中。
```

像 PUSH 一样，POP 只能操作 16 位或者 32 位操作数。不要尝试从堆栈中弹出一个数据到 8 位的寄存器中(如 AH 或 CL)。

POP 的工作方式与 PUSH 非常相似，只是方向相反。像 PUSH 一样，从堆栈中弹出多少数据取决于操作数的大小。如果要把从堆栈中弹出的数据放入一个 16 位寄存器，那么将从栈顶弹出两个字节。如果要把堆栈中弹出的数据放入一个 32 位寄存器，那么将从栈顶弹出四个字节。一定要注意，在 CPU 或者 Linux 中没有任何东西记得放在堆栈中的数据项长度是多少。记住最后一个压栈的数据项的长度由你本人负责。如果最后一个压入堆栈的数据项是一个 16 位寄存器的值，但是却要将堆栈中的数据弹出到一个 32 位寄存器中，那么将会使得弹出的数据比

压入的数据多出两个字节。在某些情况下(但是很少)你可能希望这样做，但是你当然不想是由于意外才做这件事情了！

当 POP 指令被执行时，工作顺序是这样的：首先，当前存放在 ESP 中的值所代表的地址处的数据(无论是 16 位值，还是 32 位值，取决于操作数)被从堆栈中复制下来，然后放入 POP 的操作数中，无论你指定该操作数是什么；此后，ESP 递增(而不是递减) 步长为操作数的长度，这样，实际上 ESP 在堆栈中朝着远离低内存的方向向上移动两个或者四个字节。

当压栈时，ESP 在放入一个字到堆栈之前先递减，但是当弹出数据时，在从栈顶删除了这个字之后才进行递增，这一点非常重要。某些除了 x86 世界之外的其他处理器恰好以相反的方式工作，那种方式也很好，只是不要将它们搞混。对于 x86 而言，以下语句永远为真：除非栈完全为空，否则 SP 指向的永远是真实的数据，而不是空白的空间。

一般来说，你不用刻意记这一事实，因为 PUSH 和 POP 为你全权处理这件事情，根本不需手动跟踪 ESP 指向什么位置。如果你决定直接操作堆栈指针，了解 PUSH 和 POP 背后的事件顺序(一个本书中未涵盖高级话题)将会对你有所帮助。

关于 POPA 和 POPAD 指令，一个重要的注意事项是：存储在堆栈指针寄存器中的值不会受到影响！换句话说，PUSHA 和 PUSHAD 将把当前堆栈指针寄存器中的值压入堆栈。然而，POPA 和 POPAD 却丢弃它们在堆栈上找到的堆栈指针寄存器的值，并不改变 SP/ESP 中的值。这样做很有意义：在 CPU 忙于处理堆栈工作的同时改变堆栈指针的值将会导致混乱。

图 8-3 更详细地展示了堆栈的操作过程。在程序执行的某个假设点处的 4 个 16 位“X”系列通用寄存器的值显示在图的上方。AX 第一个被压入堆栈。它的最低有效字节存放到 ESP 中，最高有效字节存放到 ESP+1 中。(记住，这两个字节被作为一个单位一次压入到堆栈中！)

每当一个 16 位寄存器被压入堆栈时，ESP 都朝着低位内存的方向向下递减两个字节。前三列分别显示了 AX，BX 和 CX 被压入堆栈的情形，但是要注意第四栏，它反映了当指令 POP DX 被执行时发生的情况。堆栈指针寄存器沿着远离低位内存的方向递增 2 个字节。DX 中现在包含了一份 CX 寄存器中的内容的拷贝。事实上，CX 被压入堆栈中，然后立即被弹出到 DX 中了。

以这种方法将 CX 中的值复制到 DX 中可能绕了个大圈。使用 MOV DX,CX 要更快些，也更直接些。不过，通过堆栈移动寄存器的值有时是必要的。请记住，MOV 指令不会操作 Flags 或者 EFlags 寄存器。如果想加载一个 Flags 或者

EFlags 的副本到某一寄存器中，必须首先使用 PUSHF 或 PUSHFD 指令将 Flags 或者 EFlags 的值压入堆栈，然后再使用 pop 指令将 Flags 或者 EFlags 的值从堆栈中弹出到选定的寄存器中。比如，要想将 Flags 寄存器中的值放入 BX 寄存器中，可通过如下所示指令完成：

```
PUSHF   ; 将 Flags 寄存器压入堆栈。
POP BX  ; ..立即将它弹出到 BX 寄存器中。
```

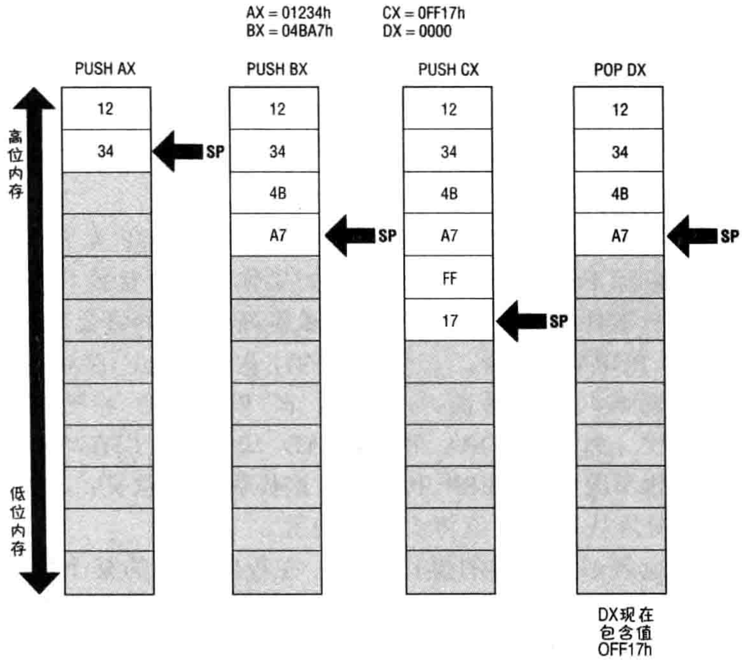


图 8-3　堆栈工作原理

并非 EFlags 中的所有位都能被 POPFD 改变。当从堆栈中弹出一个值到 EFlags 中时，VM 位和 RF 位不会受到影响。

8.2.5　临时存储

堆栈应被视为一个短期存放数据的地方。存放在堆栈中的数据项没有名字，并且一般情况下它们必须按照与进入堆栈相反的顺序弹出。记住，后进，先出。后进先出！

堆栈的一个非常好的应用是它允许数量确实太少的寄存器“身兼多职”。如果需要一个寄存器来暂时存放某些即将被 CPU 操作的值，而所有寄存器刚好都在使用中，你可以将其中一个繁忙的寄存器的值压入堆栈。在你使用该寄存器做其他事情时，它的值可非常安全地存在于堆栈之中。当你用完寄存器后，再将原来的值从堆栈中弹出——这样，在并非真的拥有一个额外寄存器的情况下，你已经获得了一个额外的寄存器的好处。(当然，代价是花费在移动那个寄存器的值到堆栈上和将其从堆栈中弹出所需的时间。我们当然不希望在需要频繁重复的循环中做这件事了！)

在程序执行过程中，短期存储是堆栈最简单、最明显的用法，但是，它最重要的用途可能是程序调用和 Linux 内核服务。现在，我们已经了解了堆栈的相关知识，可以揭开 INT 指令的神秘面纱了。

8.3 通过 INT80 使用 Linux 内核服务

eatsyscall.asm 的其他内容都是为了一条指令做铺垫，这条指令执行该程序的唯一真正工作：将一行文本显示到 Linux 控制台。这个程序的核心宗旨是：通过使用带 80h 参数的 INT 指令，实现对于 Linux 操作系统内核服务的调用。

正如第 6 章所述，操作系统是一个像神一样的东西，而且有些地方类似于巨魔，Linux 也不例外。它以神圣的方式控制着机器的所有最重要组成部分：磁盘驱动器、打印机、键盘、各种接口(以太网、USB、蓝牙等)和显示器。同时，Linux 也像一个生活在一座连接计算机的所有这些部件的桥梁下面的巨魔：你告诉巨魔你想做的事情，然后巨魔会为你代劳。

Linux 提供的服务之一是简单地(实际上，过于简单)访问个人电脑显示器。为了实现 eatsyscall.asm 程序(它只是一节为了让你体验首次编写和操作汇编语言程序的入门课程)的目标，简单的服务就足够了。

那么，如何使用 Linux 服务呢？我们必须通过 Linux 内核请求这些服务。有一种使用起来非常容易但是理解起来比较棘手的方法：通过软件中断。

8.3.1 不中断任何事情的中断

作为 x86 处理器家族的新内容，早在 1981 年，软件中断的概念就几乎让我疯掉。我瞪大眼睛，寻找中断发起者和被中断的内容，结果找了半天却发现什么也没有被中断。

这个名字不合适，尽管我承认将其称为软件中断也并非毫无道理。它们实际上是非常客气中断——当它如此客气，以至于没做任何中断时，如果你仍然将它叫作中断的话。

在 eatsyscall.asm 程序中列举过的两次真实的示例能够很好地解释软件中断和 Linux 服务的本质。正如我先前暗示的那样，Linux 让库例程(着眼于某一单一任务的机器指令序列)在它的内部隐藏起来。每个库例程指令序列都做一些有用的事情——从文件中读取数据 、发送数据到文件中、获取当前时间、访问网络端口，等等。 Linux 使用这些库例程来完成自己的工作，同时也允许你(程序员)从自己的程序中访问它们。

嗯，这里是关键问题：如何找到那些藏匿到 Linux 里面的东西呢？当然，所有的机器指令序列都拥有地址，那么为什么不公布一份所有这些有用例程的地址列表呢？

这里存在两个问题：第一，允许用户空间程序对操作系统内部进行访问是危险的。恶意软件编写者可以通过修改操作系统的关键组成部分来监视用户活动、捕获击键并转发给其他地方，等等。第二，任意给定指令序列的地址从一个安装到另一个安装会有所变化，不仅如此，甚至(随着软件不断被从个人电脑上安装、配置以及移除)从一天到另一天也会有所变化。Linux 不断在进化、完善和在现有基础上进行修复。Ubuntu Linux 操作系统版本在每年春季和秋季会发布两个主要更新，那些较小的更新则通过更新管理器定期自动下载到电脑中。修复和完善代码涉及增加、修改和删除机器指令，而这些动作会改变那些隐藏的代码序列的大小——并且作为结果，它们的位置也会发生改变。

解决方案非常巧妙。有一种调用 Linux 内部服务例程的方法，它不需依赖任何地址。大多数人称之为内核服务调用门，它是一个戒备森严的门户，位于程序运行的用户空间和 Linux 神/巨魔完成其工作的内核空间之间。调用门方案通过一个 x86 软件中断来实现。

在 x86 最开始的一段内存中，段地址和偏移地址均为 0 的地方，有一个特殊的具有 256 个数据项的查找表。每一项都是占用 4 个字节的、包括段地址和偏移地址的完整内存地址。任何 x86 机器上的第一个 1024 字节的内存被预留出来存放此表，不能在那里存放其他的代码或数据。

表中的每一个地址都被叫作中断向量。这张表作为一个整体被叫作中断向量表。每个向量都有一个 0 到 255 之间的编号。占用表中 0 到 3 字节的向量是向量 0，占用表中 4 到 7 字节的向量是向量 1，以此类推，如图 8-4 所示。

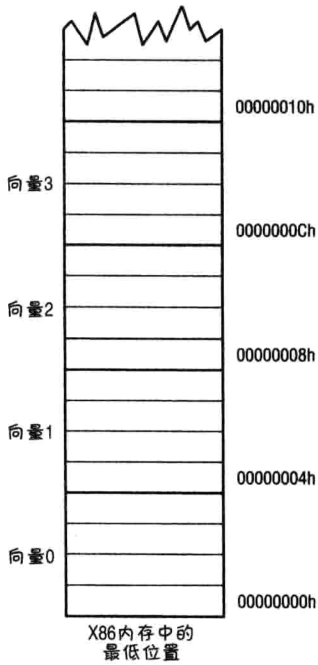


图 8-4　中断向量表

这些地址没有一个像个人电脑 BIOS 例程那样烧制到永久性内存中。当机器启动时，Linux 和 BIOS 向很多位于中断向量表中的插槽填写关于该插槽本身的中断服务程序的地址。每一个版本的 Linux 都知道其内部部件的位置，并且当升级到新的 Linux 版本时，这个新的版本会向很多位于中断向量表中的插槽填写升级之后的准确地址。

从一个 Linux 版本到另一个 Linux 版本中没有什么变化的是存放特定地址的中断编号。换句话说，从最早的 Linux 发行版本开始，中断 80h 就指向进入最漆黑的通往 Linux 服务调度程序的路线，它是一种类型的多路开关，拥有很多通往数量众多(将近 200 个)的、单独的 Linux 内核服务例程的支线。在大多数 Linux 发行和版本中，该调度程序的地址是不同的，但是不管你使用的是哪个 Linux 发行或版本，程序都能通过中断向量表中的 80h 项来访问该调度程序。

此外，程序不必非得通过查找中断向量表才能获得它们的地址。事实上，这在保护模式限制下是禁止的。该表隶属于操作系统，你甚至连去那里观察它都不可以。但是，你不一定非要直接访问表中的地址。x86 处理器包括一条机器指令，它拥有特殊的使用中断向量表的权力。eatsyscall.asm 使用 INT(中断)指令来

请求在屏幕上显示广告语字符串的 Linux 服务。eatsyscall.asm 在两个地方都有一个 INT 80h 指令。当执行一条 INT 80h 指令时，CPU 查找中断向量表，并从 80h 表项中取出地址，然后再将执行流程跳转到该地址。从用户空间到内核空间的过渡非常干净，而且完全可控。而另一边，位于储存在表中 80h 项中的地址处的调度程序开始执行，开始完成你的程序请求的服务。

这一过程如图 8-5 所示。当 Linux 在计算机引导期间被加载时，它为计算机准备使用所做的诸多事情之一就是把正确的地址放入中断向量表中的几个向量中。其中一个地址就是内核服务调度的地址，它将进入插槽 80h。

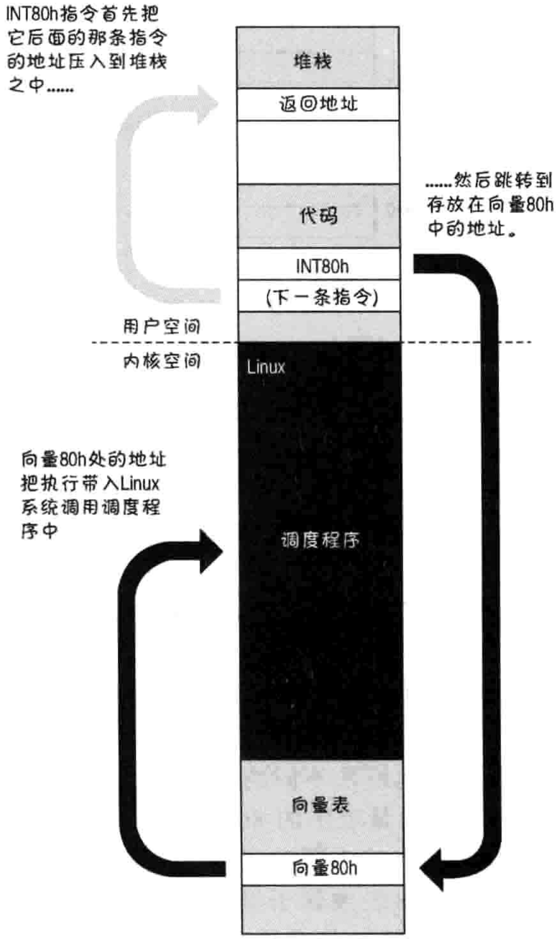


图 8-5 利用中断向量表进入 Linux

以后，当在 Linux 控制台的命令行中键入程序名称 eatsyscall 时，Linux 加载 eatsyscall 的可执行文件到用户空间内存，并允许它执行。为了访问内核服务，eatsyscall 根据需要执行 INT 80h。除了需要知道在中断向量表中的编号之外，程序不需知道任何有关 Linux 内核服务的内容。知道了那个简单的编号之后，eatsyscall 继续处于无知状态，它只是简单地让 INT 80h 指令和中断向量 80h 将它带到需要去的地方。

在芝加哥的西北侧(我长大的地方)，有一路沿着密尔沃基大道行驶的公交车。所有的芝加哥巴士路线都有一个编号，密尔沃基大道路线的编号是 56。它从纠缠不清的市中心以北的街道的某个地方出发，终点站是城区范围内的森林保护区。该森林保护区在那块森林保护林中有一个名为惠兰池的游泳池。密尔沃基大道沿线的孩子们不一定非要告诉你惠兰池的地址，他们可以以第二种方式告诉你如何到达：只需跳上 56 路公共汽车，到终点站下车即可。这与软件中断非常相像。找到能够准确指向目的地的向量的编号，乘坐该向量一直到路线的终点，不用担心蜿蜒的路线或者目的地的确切地址。

而在幕后，INT 80h 指令做了一些其他事情：在沿着向量 80h 所示的地址进入 Linux 内核之前，它把下一条指令(即紧跟在 INT 80h 指令后面的那条指令)的地址压入堆栈。像 Hansel 和 Gretel 一样，INT 80h 指令将一些“面包屑”压入堆栈中，以帮助 CPU 在“远足”到 Linux 内部之后依然能够找到返回 eatsyscall 的路线(稍后会有更多与之相关的内容)。

现在，Linux 内核服务调度程序控制着对于这 200 个单独的服务例程的访问。它是如何知道执行哪一个例程的呢？我们要告诉调度程序需要哪个服务，这可以通过将服务的编号放入寄存器 EAX 中来实现。在开始工作之前，调度程序可能还需要其他信息，并且期待你在正确的位置提供该信息(它们几乎总是在各种各样的寄存器中)。

让我们看看下面几行来自于程序 eatsyscall.asm 的代码：

```
mov eax,4           ; 指定 sys_write 系统调用
mov ebx,1           ; 指定文件描述 1: 标准输出
mov ecx,EatMsg      ; 传递消息的偏移地址
mov edx,EatLen      ; 传递消息的长度
int 80H             ; 进行系统调用，将该文本输出到标准输出
```

这个指令序列要求 Linux 显示一个文本字符串到控制台上。第一行建立了一个关键信息：需要请求的服务的编号。在本例中，它是 sys_write，服务编号为 4，它能够将数据写入一个 Linux 文件中。请记住，在 Linux 中，几乎所有东西都

是文件，其中也包括控制台。第二行告诉 Linux 写入哪个文件：标准输出。每个文件都必须有一个数字的文件描述符，前 3(0、1 和 2)个是标准的，并且永远不会改变。标准输出的文件描述符是 1。

第三行把将要显示的字符串的地址放到 ECX 寄存器中。Linux 通过它来知道我们想要显示的内容。调度程序希望我们将地址存入 ECX 寄存器中，但是那个地址只是字符串的开始位置。Linux 还需要知道字符串的长度，我们将那个值放在 EDX 寄存器中。

当内核服务编号、字符串的地址和字符串的长度被放入适当的寄存器之后，我们通过执行 INT 80h 到调度程序旅行了一圈。INT 指令是它需要的所有内容。轰！执行跨过“桥梁”进入内核空间，Linux 巨魔在那里读取 ECX 处的字符串，并且通过它那或多或少有些保密的机制，将其发送到控制台上。大多数情况下，这是一件好事儿：描述程序机制可能需要很多的信息，它就像描述你的个人生活一样一言难尽。

8.3.2 再次返回

关于进入 Linux 内部就讲这么多。那么执行如何再次返回呢？向量 80h 中的地址将执行带入了内核服务调度程序，但是 Linux 如何知道将执行传递到 eatsyscall 中的什么地方呢？软件中断的一半聪明之处是知道如何到达那里，而另一半——同样也很聪明——是知道如何再返回来。

要想继续执行在 INT 80h 指令之前面所遗留的内容，Linux 必须在一个完全可靠的地方找回返回地址，而那个完全可靠的地方不是别处，正是堆栈的栈顶。

我刚才提到(没有过多强调)，INT 80h 指令在进入一个未知地址之前，先将一个地址压入堆栈。这个地址就是下一条即将轮到执行的指令的地址：紧跟在 INT 80h 指令后面的那条指令的地址。这个位置是完全可靠的，因为，正如一台机器上只有一个中断向量表一样，在任何一个时间段中，只有一个堆栈在工作。这意味着只存在一个堆栈的栈顶——也就是 ESP 寄存器所指向的地址——通过将地址从栈顶弹出，然后跳转到那个地址，Linux 总是能够将执行送回到调用它的程序中。

这一过程如图 8-6 所示，它是图 8-5 的延续。正如 INT 指令将一个返回地址压入到堆栈中，然后跳转到存放在某一特定向量中的地址一样，有一条“配对”指令能够将返回地址从堆栈中弹出，然后跳转到这个地址。这条指令就是 IRET(用于中断返回)，它完成这一复杂却可靠的系统(在你不知道该地址的情况

下，跳转到这个地址)。再次的，其中的技巧是知道在哪里可以可靠地找到地址，在本例中，那个地方就是堆栈。

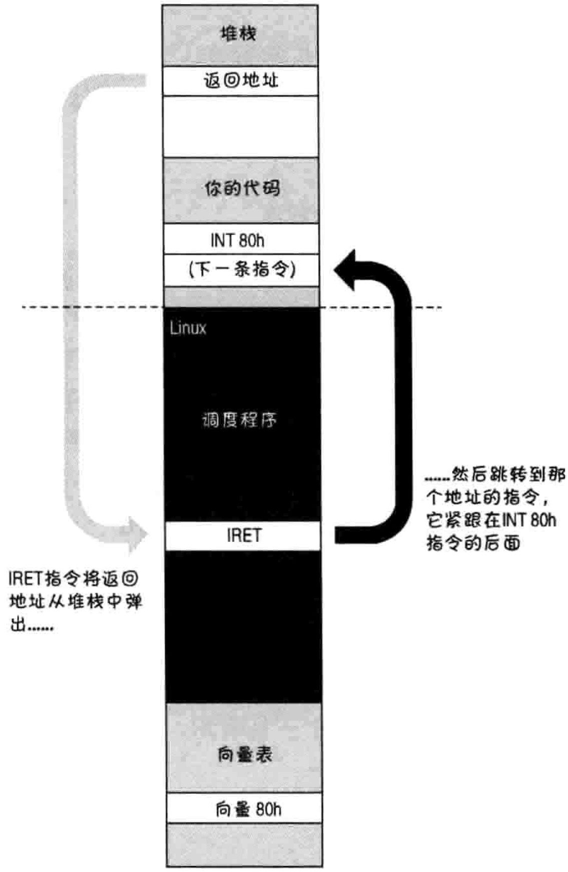


图 8-6　从中断返回

实际上，软件中断机制压入和弹出堆栈的内容不止这些，但是它的发生非常透明，我不想在这个时候加上复杂化的解释——而且你也暂时不可能编写自己的软件中断例程。那是在内核区域编程，我鼓励你追求这一目标，但是当你刚刚开始学习汇编时，它仍然是一件很遥远的事情。

8.3.3　通过使用 INT 80h 退出一个程序

在 eatsyscall.asm 中，有第二个 int 80h 指令，它完成一项谦卑却又关键的工

作：关闭程序，将控制返回给 Linux。这比它的实际情况听起来要简单，一旦你了解了更多的 Linux 内部知识以后，你将开始重视那项必不可少的、既要启动一个进程又要并关闭一个进程的工作了。

从你自己的程序的角度来看，它非常简单：将 sys_exit 服务的中断号放入 EAX 寄存器中，把返回值放入 EBX 寄存器中，然后执行 INT 80h：

```
mov eax,1     ; 指定退出系统调用
mov ebx,0     ; 将零作为返回值
int 80H       ; 进行系统调用以终止程序
```

返回值是一个数字值，但是你可以将其定义为任意你想要的内容。从技术上来讲，它的内容没有任何限制(除了必须适合于存放在 32 位寄存器中之外)，但是按照惯例，返回值为 0 意味着“一切工作都完成了; 正常关闭”。返回值为 0 以外的数字则通常表明发生了某种类型的错误。请记住，在更大一些的程序中，必须留意那些没有按预期工作的事情：找不到磁盘文件、磁盘驱动器已满，等等。如果程序不能完成其工作，必须提前终止，它应该以某种方式告诉你(或在某些情况下，另一个程序)哪里出了问题。返回值就是一个能够做到这一点的好办法。

以这种方式退出不仅仅是一件出于美观的事情。我们编写的每一个程序都必须以这种(通过内核服务调度程序来进行 sys_exit 调用的)方式退出。如果一个程序仅仅是“运行到了边缘”，那么它实际上将会结束，但是 Linux 会提交一个分段错误，我们对于发生了什么事情仍然一无所知。

8.3.4　软件中断 VS 硬件中断

你可能还是想知道为什么这样的机制被叫作“中断”，这是一个具有历史根源的合理问题。软件中断从一个确实涉及一些真正中断的老式机制演变而来：硬件中断。硬件中断是一种 CPU 注意自身之外世界的机制。

一个内置在电脑中的相当复杂的电气系统能够让电路板发送信号给 CPU。CPU 芯片上的一个的实际的金属针被电路板设备(如磁盘驱动器、控制器或一个串口板)从某一伏特电压等级变为另一等级。通过这个引脚，外部设备能够“轻拍 CPU 的肩膀”。CPU 将这种“轻拍”识别为硬件中断。像软件中断一样，硬件中断也被进行了编号，并且对于每一个中断编号，在中断向量表中都预留一个“插槽”(表项)。在这个插槽中，存放的是一个终端服务例程(ISR)的地址，该中

断服务例程执行一些与轻拍 CPU 肩膀的设备相关的事情。例如，如果该中断信号来自一个串口电路板，CPU 将会允许该串口板从它自己传输一个字符字节到 CPU 中。

硬件中断和软件中断之间的唯一不同之处是触发到中断向量表之旅的事件。在软件中断中，触发事件是软件的一部分，也就是说，一条 INT 指令。在硬件中断中，触发事件是一个应用于 CPU 芯片本身的、没有任何 INT 指令插手这一过程的电信号。当意识到有电脉冲触发中断时，CPU 自己把返回地址压入堆栈；但是，当 ISR 执行完毕，必须通过一条 IRET 指令将执行“送回家”，就像在软件中断中所做的一样。

这里所说的用于在一个软件中断调用后的“返回”机制，实际上比听起来更加普遍。在本书的后面部分，我们将开始把程序分割成过程，这些过程通过一对儿指令来进行访问：CALL 和 RET。CALL 将下一条指令的地址压入堆栈，然后跳转到一个过程；在该过程的结尾处，一条 RET 指令将该地址从堆栈的栈顶弹出，然后允许执行继续捡起 CALL 指令后面的内容。

8.3.5 INT 80h 和可移植性盲目崇拜

10 年前，在我准备两个包含在本书第二版中的 Linux 相关章节期间，我观看了有关在用户空间程序中直接将 INT 80h 访问合并到 Linux 内核调用的可取性的辩论。有两个人几乎有失身份地尖叫起来：INT 80h 调用必须由标准的 C 函数库来调用，汇编语言对内核服务的调用应该总是通过调用(能够制造 INT 80h 内核调用的)C 函数库中的例程来间接实现。

辩论的激烈表明，我们已经不再单纯从技术优势上讨论某些事情，而是已经跨入盲目崇拜的领域。我在这里提到它是因为像这样的人仍然存在，如果你在 Linux 编程界待的时间足够长，你终会遇到他们。我的建议是：如果可能的话，尽量避免这场辩论。争论它根本没有意义，而且幸运的是，自 2000 年以来，各种编写 Linux 程序的新方式呈爆炸式发展，这足以使可移植性盲目崇拜黯然失色。

但是它可以归结为：Unix 世界一直秉承这样的理念，即一个程序应该能够在不需任何更改的情况下被重新编译，并且能够正确地运行在一个不同版本的 Unix 上，甚至一个运行在完全不同的 CPU 体系结构上的 Unix 上。这仅仅是勉强可以的，只对那些用简单语言(C)编写的相对简单的程序适用，该简单语言使用计算系统所能提供的“最小公分母”子集。进入精心制作的图形用户界面(GUI)应用程序和现代的外设之后，你将面对很多拥有大量复杂的应用程序编程接口(API)

的、不兼容的软件库，以及那些本不应该存在却非常不礼貌地存在着的设备驱动程序怪癖。

再加上近年来所有这些 API 和新的、更高级的语言(比如 Python)的不断演变，你在去年的编写的程序甚至可能今年就不在同一平台上编译了，你将面对我在很多年前得出的结论：嵌入式便携性就是一个神话。我们现在的平台如此之复杂，以至于每一个应用程序都基于特定的平台。跨平台编码可以做，但是必须修改源代码，通常折中于必须通过使用条件编译来实现——从本质上来讲，条件编译就是程序里面的 IF 语句集，该语句集通过一系列传递给编译器的参数来改变源代码；如果你在为 x86 上的 Linux 进行编译，编译这些语句；如果你在为 X86 上的 BSD Unix 进行编译，编译其他语句，等等。条件编译仅仅是佩戴在残酷的底层现实上的面具：计算机是不同的。计算机系统是不断进化的。现在已经不再是 1970 年了。

我在本节讲解的 Linux 内核服务调用实际上是针对特定的 Unix 的 Linux 实现的。其他的 Unix 实现以不同的方式处理内核调用。在 Unix 操作系统的 BSD 家族中，内核服务调度程序也是通过 INT 80h 实现的，但是，参数被传递到内核的堆栈上，而不是寄存器中。我们可以辩论技术优点，无论是好是坏，但是它们是不同的，Linux 汇编程序将无法运行在 BSD Unix 下。如果这是你要研究的问题，汇编语言可能不是出路。(要想了解更多或更少的“可移植性”编码，我认为你应该学习 Python，这是一个极具魅力的非常高级的语言，并且几乎出现在所有的 Unix 实现上。)

但是，在 Linux 的发行和甚至跨年的升级中，内核服务列表本身只是改变了一点点，改变更多的主要是最近几年添加到内核的更加神秘的服务。如果在 Linux 的 x86 发布下编写的汇编代码不会同等地运行于另一个 x86 发布中，它并不是由于调用内核服务的方式造成的。

汇编语言不是也不能是可移植的。它不是为实现移植而设计的。另一方面，也不要让任何人试图说服你去做这件事情。

8.4　设计一个有价值的程序

到目前为止，我们已经知道了设计和编写能够执行重要的、甚至非常有用的工作的实用工具所需的所有相关知识。在本节中，我们将从解决问题的工程立场来面对编写一个实用程序的挑战。这里涉及的不仅仅是编写代码。它涉及问题陈

述、问题分解(将问题分解为一个个组成部分)以及(以一系列可用汇编语言程序实现的步骤和测试的方式)提供问题的解决方案。

对于本节知识，有一个特定的“鸡和蛋”问题：要想编写一个有价值的汇编程序，如果没有条件转移语句，会很困难；如果没有通过一个有价值的程序演示条件转移语句，解释这类语句会很困难。在前面的章节里，我已经谈到了一些有关转移语句的内容，直到第九章才会对它们进行详细讲解。在本节的演示程序中，我使用的转移指令非常简单；如果你对细节感到有些模糊，请阅读第 9 章，然后再返回本节，完成这个例子。

8.4.1 问题定义

多年以前，我曾在一个研发小组中编写过一个系统，该系统用于收集和验证来自世界各地的外地办事处数据，并将其发送到一个大型中央计算设施，在那里进行汇总、分析，然后用于生成状态报告。这听起来容易之极，实际上收集来自外地办事处的数据本身的确不难。使该项目变得困难的是，它涉及几个单独的而且类型不同的计算机，这些计算机以完全不同的、往往是不兼容的方式来看待数据。这个问题与数据编码问题有关，我在第六章中曾经简短地谈到过相关内容。我们必须为数据字符处理三种不同的数据编码系统。一个以某种方式解释的字符，在另一个其他系统中，并不会被解释为相同的字符。

要想将数据从一个系统转移到另一个其他类型的系统中，我们必须编写一种能够将数据编码从一个架构翻译到另一个架构的软件。其中一个架构使用了数据库管理器，出于某些即使在当时也看起来有些特殊、在今天看来依然有些不可思议的原因，它不能很好地“消化”小写字符。在我们将数据送入系统之前，必须把所有的小写字母转换成大写字母。还有一些其他的编码问题，但这是最重要的问题之一，并且因为它是一个描述和解决都很简单的问题，所以可作为学习真正的汇编语言程序设计的一个很好的首次练习。

在最高层面上，这里要解决的问题可以陈述为：将一个数据文件中的所有小写字符转换成对应的大写字符。

知道这一点以后，对问题做笔记是个不错的主意。特别要对任一提出的解决方案的限制上做些笔记。我们曾经将这些笔记叫作解决方案的“边界”，在思考解决该问题的程序期间，必须时刻记将它在心中：

- 我们将工作在 Linux 下。
- 数据存在于磁盘文件中。

- 我们事先不知道任何一个文件的大小。
- 文件大小没有最大值或最小值。
- 我们将使用 I / O 重定向技术将文件名称传递给程序。
- 所有输入文件都采用相同的编码方案。这个程序可以假设在一个文件中的字符“a”与另外一个文件中的字符“a”所采用的编码方式相同。(在我们的例子中，这是 ASCII 码编码方式。)
- 我们必须以原来的形式保留原始文件，而不是从原始文件读出数据，然后将其再写回到原始文件中。(这是因为如果进程崩溃，我们会在没有完全生成输出文件的情况下破坏原有文件。)

在真正的项目中，可能会有很多这种笔记页，但在这里，只是几页就足以帮助我们塑造对于字符大小写问题的简单解决方案了。请注意，这些笔记扩展到必须做哪些事情，并在一定程度上对最终解决方案的性质加以限制，但是并不试图说明它必须以何种方式完成。这是我们下一步要做的事情。

8.4.2 从伪代码开始

一旦已经尽可能详细地了解了问题的性质，我们就可以开始制作解决方案了。首先，这非常类似于我在第一章中描述的过程，在那一章里，有人做出一个“即将要做” 的任务列表，用于运行一天的差事。我们应该以一种非常概括的形式陈述一个解决方案，而且应该使用尽可能少的语言。然后，通过将较大步骤分解成其包含的较小步骤来一点一点地完善这个已经陈述的解决方案。

在我们的例子里，解决方案非常容易采用比较概括的术语来陈述。让我们开工吧！这里是一种可能采用的陈述形式：

- 从输入文件读入一个字符。
- 将该字符转换成大写字符(如果需要的话)。
- 将字符写入到输出文件中。
- 重复该过程直至任务完成。

如果从一个极高的视角来看，这确实是一个解决方案。它虽然缺少细节，但并不缺少功能。如果执行列出的步骤，我们将拥有一个程序，该程序能够完成我们需要它做的事情。应该注意的是，前面的陈述并不是使用任何编程语言编写的。它们当然更不是汇编语言指令。它们是几个行为的描述，独立于任何完成这些行为的特定系统。类似这样的陈述列表，由于它们故意不被写成为特定编程环境而编写的代码，因而被叫作伪代码。

8.4.3 连续改进

从第一个完整但不够详细的解决方案陈述出发，让我们迈向另一个更为详尽的陈述。这一过程可以通过改进伪代码语句(使其更为具体地陈述那些被描述行为将如何完成)来实现。重复这个过程，每次增加更多的细节，直到我们可以很容易地将其转化为实际的汇编语言指令为止。这个过程被称为连续改进，它并不是汇编语言专用的。该过程在所有的编程语言中都有不同程度的使用，但是在汇编中运用得尤为出色。

让我们锁定上面给出的伪代码，并为其创建一个追加了细节的新版本。我们知道，我们将在 Linux 下使用该程序——这是规范的一部分，所有解决方案的边界之一——所以我们可以开始添加一些针对 Linux 环境下完成这些事情的方式的细节。下一轮改进可能如下所示：

- 从标准输入(stdin)中读入一个字符。
- 测试该字符以判断其是否为小写字符。
- 如果该字符为小写字符，则将其通过与 20h 相减来将其转换为大写字符。
- 将字符写入标准输出(stdout)。
- 重复该过程直到任务完成。
- 通过调用 sys_exit 来退出程序

在每一轮的改进过程中，我们都需要花些时间努力查看每一个行为陈述，看看有没有可能隐藏的细节，并在下一轮改进中扩大这些细节。有时这非常容易，有时则没那么简单。在前面的版本中，“重复该过程直到完成” 这句话乍听起来很明白，也很显而易见，直到你开始考虑这里的“完成”指的是什么：将数据输入到输出文件中。我们如何知道输入文件已经没有字符？这可能需要一些研究，但是在大多数操作系统(包括 Linux)中，你调用的用于从文件中读取数据的例程都会有一个返回值。这个值可以表明一次成功的读取、一次错误的读取或者某种特例结果，例如：文件是否结束(EOF)。一个扩展后(稍做重组)的解决方案版本的伪代码可能像这样：

- 从标准输入(stdin)读入一个字符。
- 测试一下是否已经到达了文件的结尾(EOF)。
- 如果已经到达了文件的结尾，我们已经完成任务，所以跳转到 exit。
- 测试该字符以判断其是否为小写字符。

- 如果该字符为小写字符，则通过将其与 20h 相减来转换为大写字符。
- 将字符写入标准输出(stdout)。
- 回去读取另一个字符。
- 通过调用 sys_exit 来退出程序。

我们就按照这种思路来进行改进工作，每次增加一些细节。请注意，这已经开始有点像现在的程序代码了。情况将会变成这样：随着语句个数的增加，对那些代表跳转目标的语句添加标号会给我们带来帮助，这样可避免混淆跳转目标，即便是在伪代码中。通过将相关语句组合在一起，它也有助于我们将伪代码分解成子块。我们迟早会得到与以下内容类似的结果：

```
Read: 设置用于 sys_read 内核调用的寄存器。
      调用 sys_read 从标准输入中读入一个字符。
      测试 EOF.
      If 我们到了 EOF， 跳转到 Exit.
      测试该字符以判断其是否为小写字符。
      If 它不是小写字符，跳转到 Write.
      通过将字符与 20h 相减来将其转换为大写字符。
Write: 为写内核调用设置相关寄存器。
       调用 sys_write 向标准输出写数据。
       跳回到标号 Read 处，读取另一个字符。
Exit: 设置通过 sys_exit 来停止程序所需的寄存器
      调用 sys_exit。
```

这是一个将伪代码语句“转到”操作系统和即将使用的编程语言方向的很好示例。所有编程语言都有它们的怪癖、局限性和一般的“形状”(语法特点)，在生成伪代码的过程中，如果记住了这个“形状”，会使过渡到真正代码的最后环节会变得更容易。

某些时候，伪代码拥有所有它可以包含的、仍然为伪代码的细节。要想更进一步，你将不得不开始将伪代码转换为真正的汇编代码。这意味着你必须取出每一条陈述语句，并问自己：我是否知道如何将这种伪代码转换成一条或多条汇编语言语句？当你还是初学者的时候，这样做是正确的；但是，即使已经赢得了一枚汇编语言程序员的印章，你也不可能知道所有即将被了解的一切。在大多数编程语言(包括汇编)中，有时实现某一特定行为往往存在几种不同的方法。有些方法可能比其他方法快一些；有些方法虽然可能慢一些，但却更易于阅读和修改。有些解决方案可能仅限于 x86 系列处理器的子集中。你的程序需要运行在古老的 386 或者 486 CPU 上吗？或者你能假设每个人都至少拥有一个奔腾处理器吗？

(对于最初问题的任一可用解决方案，原始注释表中的说明都应包含这些边界条件。)

从伪跳代码“跳”到指令似乎跨度很大，但是好消息是，一旦已经将伪代码转换为指令，就能将该文本变为一个汇编语言源代码文件，并且可让 NASM 大致检查一下，以找出语法错误。估计你要花些时间修复汇编错误，然后是程序漏洞，但是如果在完善过程中，你能保持头脑清醒且耐心十足，你可能会惊讶于在自己的第一次尝试中，程序居然运行得如此之好了。

清单 8-1 展示了前述伪代码的一个比较胜任的真正的汇编翻译。通读它，看看你能否领会从伪代码翻译过来的内容，进而了解一下你现在对汇编语言掌握了多少。但是，它并没有在任何真正意义上“完成”。它只是在连续改进以获得真正代码的过程中的“首次切割”。 首先，它需要更好的注释，——但是比什么都重要的是，它需要一些努力的思考：对于最初的问题而言，它是一个好到什么程度的解决方案？它的完整性如何？注意，一个能够工作的程序不一定是一个完整的程序。

清单 8-1：uppercaser1.asm

```
section .bss
        Buff resb 1

section .data

section .text
        global _start

_start:
        nop             ; 这个无操作指令使调试器感到非常高兴。

Read:   mov eax,3       ; 指定 sys_read 调用。
        mov ebx,0       ; 指定文件描述符 0: 标准输入。
        mov ecx,Buff    ; 传递即将从中读取数据的缓冲区的地址
        mov edx,1       ; 告诉 sys_read 从标准输入读入一个字符。
        int 80h         ; 调用 sys_read
        cmp eax,0       ; 观察 sys_read 在 EAX 寄存器中的返回值。
        je Exit         ; 如果为零(意思是 EOF)则跳转到 Exit。
                        ; 或者不为零，则测试它是否为一个小写字母。
        cmp byte [Buff],61h  ; 将输入字符与小写字母‘a’相比较
        jb Write             ; 如果在 ASCII 表中比 a 小 ，则不是小写字符。
```

```
        cmp byte [Buff],7Ah    ; 测试输入字符与小写字符 'z' 比较。
        ja Write               ; 如果在 ASCII 表中比 'z' 大，则不是小写字符。
                               ; 此时，我们已经拥有了一个小写字符
        sub byte [Buff],20h   ; 从小写字符中减去 20h，得出相应的大写字符...
                               ; ......然后将该字符写出到标准输出。

Write:  mov eax,4              ; 指定 sys_write 调用。
        mov ebx,1              ; 指定文件描述符 1：标准输出。
        mov ecx,Buff           ; 传递要写出的字符的地址。
        mov edx,1              ; 传递要写出的字符的数量。
        int 80h                ; 调用 sys_write...
        jmp Read               ; ......然后跳转到开始处，获取另外一个字符。

Exit:   mov eax,1              ; 退出系统调用的代码
        mov ebx,0              ; 将零作为返回值而返回 Linux。
        int 80H                ; 进行内核调用以退出程序。
```

这看起来有点吓人，但是它几乎包含了我们已经讨论过的全部指令和概念。有些注释语句此时你可能还不能完全理解：

- Buff 是一个尚未初始化的变量，因此位于程序的.bss 段中。Buff 没有初始值，并且不包含任何内容，直到我们从 stdin 中读入一个字符并将其储存在那里。
- 当 sys_read 调用返回值为 0 时，说明 sys_read 已经到了从中读取数据的文件的结尾处。如果返回一个正数，那么这个值表示它已从文件中读出的字符的个数。在本例中，由于我们只要求一个字符，所以 sys_read 或者返回一个 1 以表明计数值；或者返回一个 0 以表明我们已经完成字符处理。
- CMP 指令比较它的两个操作数，并设置相应的标志。CMP 指令后面的条件转移指令根据标志的状态采取行动。(下一章将涉及更多与此相关的内容。)
- 如果前面的 CMP 指令的左操作数小于右操作数，JB(如果在下面则跳转)指令跳转。
- 如果前面的 CMP 指令的左操作数大于于右操作数，JA(如果在上面则跳转)指令跳转
- 由于内存地址(如 Buff)只是指出内存中的一个没有指明固定大小的位置，所以必须在 CMP 和它的内存操作数之间放一个限定符 BYTE，告诉

NASM，它要比较两个 8 位的值。在本例中，这两个 8 位值是一个 ASCII 字符(如“w”)和一个十六进制值(如 7Ah)。

可以通过 I/O 重定向来运行可执行程序。运行 uppercaser1 的命令行如下所示：

```
./uppercaser1 > outputfile < inputfile
```

“outputfile”和“inputfile”可以是任何文本文件。请尝试一下以下命令行：

```
./uppercaser1 > allupper.txt < uppercaser1.asm
```

当该程序运行时，将创建一个 allupper.txt 文件，并将程序源代码的所有字符强制转换为大写之后写入该文件。

8.4.4　不可避免的“哎呀！”时刻

在尝试这一将伪代码转换为机器指令的最后步骤时，你可能会发现误解或者忘记了某些内容，你的伪代码并不完整，或者并不正确(或者既不完整，也不正确！)，特别是当你还是初学者时。你可能还会认识到，有一些用汇编语言实现的、更好的完成某事的办法，它能比直接从字面翻译伪代码带来更好的效果。学无止境，无论认为自己学得多么棒，你都将永远处于学习状态。

一个很好的例子，同时也是在你阅读前述汇编代码过程中真正有可能发生的事情是：程序没有检测到错误。它假设用户为 I/O 重定向输入的输入文件已经存在并且没有损坏，文件里面已经存有数据，在当前用于输出文件的驱动器上也会有空间，等等。这种操作方式很危险，即使它能够侥幸顺利完成。与文件相关的 Linux 系统调用都能返回错误值，任何使用它们的程序都应检查这些错误值，并据此采取相应的行动。

总之，在整个过程中，总要上演几次你必须认真推敲伪代码，甚至将其完全推翻，然后再重新开始的场景。当你处于将伪代码转换为机器指令的最后阶段时，这种(令人讨厌的)情况经常发生。请做好思想准备。

如果你了解一些关于低级文件输入/输出的相关知识，在你身上还可能发生一个问题：Linux 的 sys_read 内核调用不仅限于一次返回一个单个字符。你将一个缓冲区的地址传递到 sys_read 中，sys_read 将尝试从输入文件中向该缓冲区填充字符，你告诉它填充多少它就填充多少。如果设置了一个 500 字节大小的缓冲区，你能够让 sys_read 将 500 个字符从标准输入中读出，然后放入该缓冲区中。因此，对 sys_read 的一次简单调用就能够一下子带给你 500 个字符(或 1000，或

16 000)来处理。这样可以减少 Linux 在切换文件系统和你的程序上所花费的时间，但是它也大大改变了程序的结构。你先填充一个缓冲区，然后必须步入该缓冲区，一次取出一个字符，把所有小写字符的内容转换为大写字符。

在改进问题解决方案的伪代码过程中，你应该已经知道了前期步骤，——在这一阶段待了一段时间之后，你终将知道。有很多你的大脑和指尖都需要的诸如此类的细节，你不可能一个下午就将它们变为永不磨灭的记忆。这种启示经常迫使你“备份”一个或两个迭代，并且重新改写某些伪代码。

8.4.5 扫描缓冲区

我们当前正在研究的例子就遇到了这种情况。程序需要一些错误处理，在本例中，这些错误处理主要涉及检查 sys_read 和 sys_write 的返回值，并在 Linux 控制台上显示一些有意义的信息。显示出错信息和显示为廉价小吃店食客准备的广告语之间并没有技术上的区别，所以你可以自己添加一些错误处理作为练习。

然而，更为有趣的挑战涉及缓冲文件输入/输出。Unix 的读写内核调用是面向缓冲区而非面向字符的，因此我们必须重新修改伪代码以填充字符到缓冲区中，然后再处理这些缓冲区。

让我们再次回到伪代码，并进行以下尝试：

```
Read:为 sys_read 内核调用设置寄存器。
     调用 sys_read 来从标准输入中读入。满满一缓冲区的字符。
     测试 EOF。
     If 我们到了文件的结尾，跳转到 Exit.

     将寄存器设置成一个扫描该缓冲区的指针。
Scan: 测试缓冲区指针处的字符，判断它是否为小写字符。
     If 它不是一个小写字符，跳过转换。
     通过将字符与 20h 相减来将其转换为对应的大写字符。
     递减缓冲区指针。
     If 缓冲区中仍然有字符，跳转到标号 Scan 处。

Write: 为写内核调用设置寄存器。
      调用 sys_write 将处理后的缓冲区内容写出到标准输出。
      跳回标号 Read 处，然后获取另一个充满字符的缓冲区。

Exit: 为通过 sys_exit 来终止程序而设置寄存器。
      调用 sys_exit。
```

这里添加了从磁盘读取缓冲区、扫描和转换缓冲区中的字符、以及后来将其写回磁盘缓冲区所需要的一切内容。(当然，缓冲区必须从一个字符扩大到某些有用的大小，如 1024 个字符。）缓冲区的技巧要点是建立一个指向缓冲区的指针，然后检查(如有必要)和转换指针所代表的地址处的字符。然后，将指针移到缓冲区中的下一个字符，做同样的事情，重复该过程，直到已经处理完缓冲区中的所有字符。

扫描缓冲区是一个很好的关于汇编语言循环的例子。在每次执行循环时，我们都必须测试某些东西，以确定是否已经完成任务并退出循环。本例中，“某些东西”就是指针。我们可以将指针指向缓冲区的开始处，然后不断测试，以判断它是否到了缓冲区的结尾；或者我们也可以设置该指针使其指向缓冲区的结尾处，从后往前工作，然后不断测试，以判断它是否到了缓冲区的开始处。

这两种办法都可以工作，但是从缓冲区尾部开始，然后从后往缓冲区的开始处工作的方式速度可以更快一点，使用的指令也相对少一些。(稍后将对其进行讲解。)我们的下一轮改进应该开始谈论具体内容：哪个寄存器该做哪些事情，等等：

```
Read: 为 sys_read 内核调用设置寄存器。
      调用 sys_read 从标准输入中读入满满一缓冲区的字符。
      将读入字符的数量存放到 esi 寄存器中。
      测试 EOF (eax = 0)。
      If 我们到了文件的结尾，跳转到 Exit。

      将缓冲区地址放入 ebp 中。
      将读入缓冲区中的字符的数量放入 ecx 寄存器中。
Scan: 将 [ebp+ecx]处的字节与‘a’相比较。
      If 该字节在 ASCII 表中的顺序比‘a’小，跳转到标号 Next 处。
      将 [ebp+ecx]处的字节与‘z’相比较。
      If 该字节在 ASCII 表中的顺序上比‘z’大，跳转到标号 Next 处。
      通过将该字节与 20h 相减来将其转换为相应的大写字符。

Next:将 ecx 寄存器递减 1。
      If 不为零，跳转到标号 Scan。

Write: 为写内核调用设置寄存器。
      调用 sys_write 将处理后的缓冲区内容写出到标准输出。
      跳回标号 Read 处，然后获取另一个充满字符的缓冲区。

Exit: 为通过 sys_exit 来终止程序而设置寄存器。
```

调用 sys_exit。

从这个改进中可以看到，需要执行的测试不止一个，而是两个。小写字母代表了 ASCII 序列中的一个范围，范围要有开头和结尾。我们必须确定受检查的字符是否在这个范围之内。要想做到这一点，需要测试这个字符，看看它是否低于小写字母范围的最小值(“a”)，或者高于小写字符范围的最高字符(“z”)。如果正在测试的字符不是小写字符，那么不需要做任何处理，我们直接跳转到指向下一个字符的代码处。

在缓冲区内部“导航”涉及两个寄存器。缓冲区的开始地址被放入 EBP 寄存器中。缓冲区中的字符数被放入 ECX 寄存器中。如果把这两个寄存器相加，将得到缓冲区中的最后一个字符的地址。如果将 ECX 寄存器中的字符计数值递减，EBP 和 ECX 之和将指向缓冲区的倒数第二个字符。每次递减 ECX，你都会得到一个与缓冲区的开始位置更近一些的字符的地址。当 ECX 递减为零时，你将到达缓冲区的开始位置，所有字符处理完毕。

8.4.6　缓冲溢出(“Off By One”)错误

但是，请等一会儿……不完全正确。在伪代码中有一个漏洞(bug)，它是所有汇编语言初学者最常见的错误之一：传说中的缓冲溢出(“off by one”)错误。EBP 和 ECX 之和将指向一个超出缓冲区结尾的地址。当 ECX 中的计数值为零时，仍然有一个字符(在缓冲区的最开始位置的那个)没有受到检验，并且保持(如果是小写)不变。解释这个漏洞来源的最简单方法就是将其画出来，如图 8-7 所示。

在本书的存档文件中有一个很短的文本文件，名为 gazabo.txt。它只包含一个毫无意义的单词“gazabo”和一个 EOL 标记，一共 7 个字符。图 8-7 显示了 gazabo.txt 文件被 Linux 加载到一个内存缓冲区之后的样子。缓冲区的地址已经加载到寄存器 EBP 中，字符数(在这里，是 7)加载到 ECX。如果你将 EBP 和 ECX 相加，由此产生的地址将超过缓冲区的结尾，进入一个尚未使用的(但愿如此！)内存中。

每当地址偏移和某种计数值合二为一的时候，这种问题都有可能发生。计数从 1 开始，而偏移量从 0 开始。字符#1 实际上位于相对于缓冲区开始位置偏移地址为 0 的地方，字符#2 在偏移地址为 1 的地方，等等。我们正在尝试将一个 ECX 中的值既作为计数值来用又作为偏移地址来用，如果缓冲区里面的偏移地址

假定从 0 开始，一个缓冲溢出错误将不可避免。

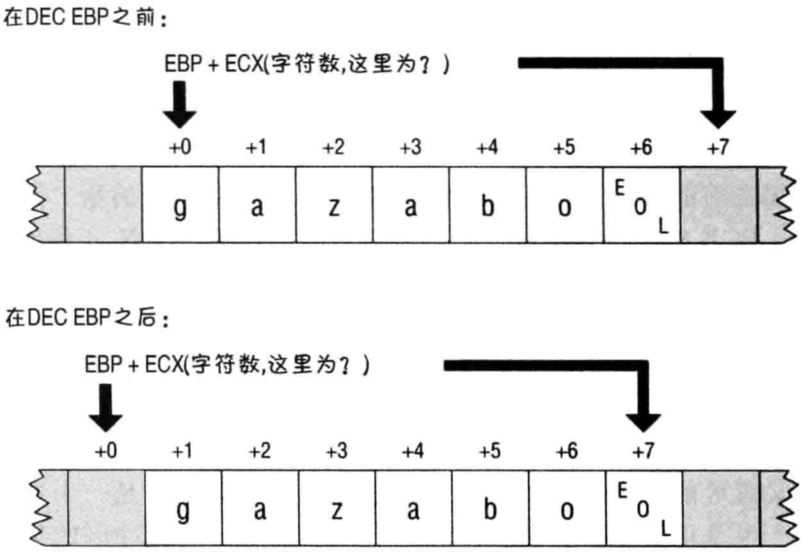


图 8-7　缓冲溢出错误

解决的办法很简单：在开始扫描前，将缓冲区的地址(它被存储在 EBP 中)递减 1。EBP 现在指向缓冲区的第一个字符前面的内存位置。EBP 以这种方式设置之后，我们就可以将 ECX 中的值既作为一个计数值来用，又作为一个偏移地址来用了。当 ECX 中的值递减为 0 时，我们已经处理了“g”字符，并且退出循环。

下面，我将把那个可怕的跳转变为实际的机器指令，但是为了简洁起见，这里将只显示循环本身：

```
; 设置寄存器，用于转换缓冲区这一步骤：
      mov ecx,esi                ; 把读入的字节数放入 ecx 寄存器中。
      mov ebp,Buff               ; 把缓冲区的地址放入 ebp 寄存器中。
      dec ebp                    ; 把缓冲区的地址调整 1 个字节

; 搜查该缓冲区，并将小写字符转换成大写字符：
Scan:
      cmp byte [ebp+ecx],61h     ; 测试输入字符，将其与小写字符 'a' 相比较
      jb Next                    ; 如果 ASCII 码值比 'a'的 ASCII 值小，
                                   则不是小写字符
      cmp byte [ebp+ecx],7Ah     ; 测试输入字符，将其与小写字符'z'相比较
```

```
        ja Next                        ; 如果ASCII码值比'z'的ASCII码值大，
                                         则不是小写字符。
; 此时，我们有了一个小写字符。
        sub byte [ebp+ecx],20h         ; 与20h相减，得出大写字符……
Next: dec ecx
        jnz Scan
```

开始扫描之前的缓冲区状态和指针寄存器显示在图 8-7 的第二部分中。在第一轮循环中，ECX 中的值是缓冲区中的字符计数。EBP+ ECX 之和指向缓冲区结尾处的 EOL 字符。在下一轮循环中，ECX 被递减为 6，EBP + ECX 指向“gazabo”中的“o”。在每轮循环中，我们递减 ECX，并通过 JNZ 指令查看零标志，当零标志未被设置时，该指令跳回到标号 Scan 处。在最后一次循环中，ECX 中包含的值为 1，EBP + ECX 指向缓冲区中第一个位置处的“g”。只有当 ECX 被递减为零时，JNZ 才“失败”，循环才结束。

纯粹主义者可能认为，在循环开始前递减 EBP 中的地址是一个冒险的黑客行为。他们有一半是正确的：递减后，EBP 将指向内存中的缓冲区的边界以外的位置。如果程序试图向该位置写入一个值，则可能破坏另一个变量，或者导致一个分段错误的结果。这里，循环的逻辑并不要求在那个特定的地址写入数据，但是很容易误操作。

处理 off-by-one 错误的“合适”方法来就是让 EBP 一直指向缓冲区的真正开始位置，然后在循环开始时而不是在结束时递减 ECX。测试 ECX，判断其是否为零的工作仍然必须要做，但是要在循环的最后，通过一条单独的 CMP 指令来完成。这样不仅能够正常工作，而且指针总是指向缓冲区内部的内存位置：

```
; 设置寄存器，用于转换缓冲区这一步骤：
    mov ecx,esi                  ; 把读入的字节数放入ecx寄存器中。
    mov ebp,Buff                 ; 把缓冲区的地址放入ebp寄存器中。

; 搜查该缓冲区，并将小写字符转换成大写字符：
Scan:
    dec ecx                      ; 递减字符计数器
    cmp byte [ebp+ecx],61h       ; 测试输入字符，将其与小写字符 'a' 相比较
    jb Next                      ; 如果ASCII码值比 'a'的ASCII码值小，则不
                                   是小写字符
    cmp byte [ebp+ecx],7Ah       ; 测试输入字符，将其与小写字符 'z' 相比较
    ja Next                      ; 如果ASCII码值比'z'的ASCII码值大，则不
                                   是小写字符。
```

```
                                        ; 此时，我们拥有了一个小写字符。
   sub byte [ebp+ecx],20h               ; 与 20h 相减，得出大写字符……
Next: cmp ecx,0                         ; 判断字符计数器的值是否为 0
   jnz Scan                             ; 如果没有，则跳转回去，继续循环
```

但是，这样做是需要付出代价的：循环内部的指令比以前多出了一条。当你只打算进行较少次数的循环时，它并没有多大影响，但是，通过在循环内部使用除了绝对必要的指令之外不采用任何多余的指令以使循环尽可能紧凑，这是一种非常好的实践。如果循环需要运行数千、数万或者数百万次的话，即使是非常微小的时间片累加起来，执行速度也能明显变慢。

将所有伪代码全部转换为汇编代码之后的完整程序如清单 8-2 所示。

清单 8-2：uppercaser2.asm

```
; 可执行程序名 : uppercaser2
; 版本 : 1.0
; 创建日期 : 3/25/2009
; 最后修改日期 : 3/25/2009
; 作者 : Jeff Duntemann
; 描述: 一个为 Linux 编写的简单汇编程序，使用 NASM 2.05,
;       展示了简单的文本文件输入/输出 (通过重定向),
;       为了实现从输入文件读入一块缓冲区，强制将小写字符转换位为大写,
;       并将修改过的缓冲区写入到一个输出文件。
;
; 以这种方式来运行它:
; uppercaser2 > (output file) < (input file)
;
; 通过使用这些命令来生成它:
; nasm -f elf -g -F stabs uppercaser2.asm
; ld -o uppercaser2 uppercaser2.o
;
SECTION .bss                ; 包含未初始化数据的段

    BUFFLEN equu 1024       ; 缓冲区的长度
    Buff: resb BUFFLEN      ; 缓冲区文本本身

SECTION .data               ; 包含已初始化数据的段

SECTION .text               ; 包含代码的段
```

```
global _start               ; 连接器需要这个来找到入口点！

_start:
    nop                     ; 这个无操作指令让 gdb 感到非常高兴……

; 从标准输入中读入满满一缓冲区文本：
read:
    mov eax,3               ; 指定 sys_read 调用
    mov ebx,0               ; 指定文件描述符 0：标准输入
    mov ecx,Buff            ; 传递要从中读入数据的缓冲区的偏移地址
    mov edx,BUFFLEN         ; 传递一次循环中要读入的字节数
    int 80h                 ; 调用 sys_read 来填充缓冲区
    mov esi,eax             ; 复制 sys_read 的返回值并妥善保管
    cmp eax,0               ; 如果 eax=0,则 sys_read 到了标准输入的结尾
    je Done                 ; 如果(与 0 相比较)相等则跳转

; 设置寄存器，用于处理缓冲区这个步骤：
    mov ecx,esi             ; 将读入的字节数放入 ecx 寄存器中
    mov ebp,Buff            ; 将缓冲区的地址放入 ebp 寄存器中
    dec ebp                 ; 调整对偏移地址的计数值

; 详细检查缓冲区，并将小写字符转换成大写字符：
Scan:
    cmp byte [ebp+ecx],61h      ; 测试输入字符，将其与小写字符 'a' 相比较
    jb Next                     ; 如果 ASCII 码值比 'a'的 ASCII 码值小，则
                                  不是小写字符
    cmp byte [ebp+ecx],7Ah      ; 测试输入字符，将其与小写字符 'z' 相比较
    ja Next                     ; 如果 ASCII 码值比'z'的 ASCII 码值大，则不
                                  是小写字符
  ; 此时，我们拥有了一个小写字符。
    sub byte [ebp+ecx],20h      ; 与 20h 相减，得出大写字符……
Next:  dec ecx                  ; 递减字符计数器
    jnz Scan                    ; 如果仍然有字符，则继续循环

;将满满一缓冲区处理过的文本写出到标准输出：
Write:
    mov eax,4                   ; 指定 sys_write 调用
    mov ebx,1                   ; 指定文件描述符 1：标准输出
    mov ecx,Buff                ; 传递缓冲区的偏移地址
    mov edx,esi                 ; 传递缓冲区数据的#个字节
    int 80h                     ; 进行 sys_write 内核调用
```

```
    jmp read                    ; 回去继续循环，加载另一满满的缓冲区数据
; 全部搞定！让我们结束这个“晚会”：

Done:
    mov eax,1                   ; 退出 Syscall 的代码
    mov ebx,0                   ; 将零作为返回值而返回
    int 80H                     ; 进行 sys_exit 内核调用
```

8.4.7 进一步学习

无论使用什么语言编程，这个一般过程都会很好地为你服务。当你在这个项目以及所有未来项目上继续前行时，这里是一些需要牢记的事项：

- 请记住，没有人说过你必须一次性将所有内容从伪代码转换为机器指令。是的，连续改进应该是连续的。一个完全合理的问题陈述可能既包括指令也包含伪代码。随着时间的推移，你会不断发展适合于自己的技术，随着你逐渐变成一名越来越自信的程序员，你所需要的完善次数将会越来越少，而得到的结果则会越来越好。
- 不要害怕画图。当你试图处理一个复杂的循环或者任何需要处理许多环节的过程时，潦草地画在方格信笺上的铅笔素描指针、缓冲区等等，可以为你带来很大的帮助。
- 保存你的笔记，无论它有多么丑陋。编程过程中的记忆会变得陈旧。如果你编写了一个实用工具，并且使用了它 6 个月，在试图重新改进它之前，你可能需要对它的内部工作机制进行一下复习。将所有相关内容都存放到一个文件夹里，包括写入磁盘文件中的伪代码的打印结果。

我们在本章中开发的程序是一个简单的 Unix 文本过滤器的例子。过滤器在 Unix 工作中十分常见，我们在后面的章节中还将重复这一概念。同时，请返回 uppercaser 程序并为其添加错误检查，无论是在读的方面，还是在写的方面。你可能需要查阅一些有关 Linux 系统调用的参考资料，那也是一种很好的实践体验。“研究”可能是编程过程中最为棘手的一个环节，而且它永远也不会变得越来越容易；请相信我。

第 9 章

位、标志、分支和表

轻松进入主流汇编编码

到目前为止，大家已经看到，我解释问题的一般方法都是自顶开始思考，然后再向下到细节。之所以这样做，是因为人类就是采用这种方法来学习的：将某一单独事实(知识点)添加到一个能够清晰表明其中各个事实(知识点)如何相互关联的较大框架中。从细节到整体的思考方法是可行的，但是在我五十六年来对各种学科知识的求索过程中，有一点变得非常清晰，那就是：先准备一个全面的框架会使建立各种事实(知识点)之间的联系变得更容易。就像要把好多石头铲到一个箱子里之前，先将它们仔细地码成一个整齐的堆一样，如果目标是把这些石头放进一个箱子，那么，在开始捡起这些石头之前，先将其各就各位会更好些。

这里也一样。大图(即框架)已经基本就位了，从书中这一部分开始，我们将研究一下汇编代码的细节，看看它们是怎么样融入这幅大的“风景”的。

9.1 位就是二进制位(字节也是二进制位)

汇编语言对位情有独钟。

毕竟，字节是由位构成的，汇编语言的基本技能之一就是创建字节以及将这些字节拆开。“位映射”(bit mapping)技术在汇编语言中使用十分广泛。位映射对字节中每一位赋予特定的含义以节省空间，进而从给定大小的内存空间中挤出剩余位，作为他用。

在 X86 指令集里，有一族指令能够让我们通过应用按位布尔逻辑运算来操作字节中的位。以下就是这些聪明的逻辑指令：AND、OR、XOR 和 NOT。另外一族指令能够对单个字节或者字进行移位操作，以下就是经常用到的移位/循环移位(shift/rotate)指令：ROL、ROR、RCL、RCR、SHL 和 SHR。(还有一些其他的指

令，本书不再一一介绍。)

9.1.1　位编号

位处理需要我们以某种方式指出所处理的是哪一位。根据约定，在汇编语言中，位被从 0 开始编号，通常从字节、字或者正在以位映射方式使用的其他项的最不重要的位开始。所谓最不重要的位，是指在二进制数制系统中拥有最小值的那个位。如果以传统的方式书写一个以二进制表示的数值，它通常是最右边的那个位。(注：最不重要的位也叫作最低有效位，最重要的位也叫作最高有效位。)

图 9-1 以 16 位字为例进行了展示。位编号的工作方式始终一样，不管处理的位有多少个：字节，字，双字，或更长的字。第 0 位通常在最右端，然后依次往左，位编号递增。

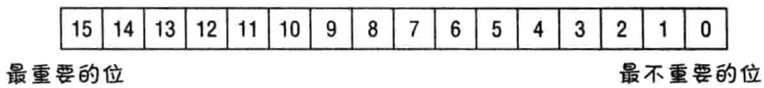


图 9-1　位编号

当你数这些二进制位的时候，从最右边的那个位开始，并且从零开始进行编号。

9.1.2　逻辑操作

布尔运算听起来挺神秘，也挺可怕，但是它却非常明显地反映了一般性思维和行为。例如，我们在日常生活中进行决策的时候，经常要用到布尔操作符与(AND)。比如，要想填写一张不被拒付的支票，支票账户必须得有钱，并且需要在支票本上进行登记。二者缺一不可。你不能填写一张不存在的支票，也不能填写一张支付账户上没钱的支票。那些需要用到支票本的人们，必须经常使用与(AND)操作符。

当数学家谈到布尔逻辑的时候，他们操作名为真或者假的抽象值。逻辑与操作符工作原理如下：条件 1 与条件 2，当且仅当二者同时为真时结果为真；如果二者之一为假，结果就为假。

实际上，这两个输入值有 4 种不同的组合，所以在这两个值之间的逻辑操作通常用真值表来进行总结。逻辑与的真值表如表 9-1 所示。

表 9-1　形式逻辑中的与真值表

条件一	操作符	条件二	结果
假	AND	假	假
假	AND	真	假
真	AND	假	假
真	AND	真	真

真值表理解起来并不困难。它是与操作符应用于两个输入条件的所有可能结果的总结。对于与操作符，最重要的是要记住，只有两个输入值都为真的时候，结果才为真。

这就是数学家们眼中的与操作。在汇编术语中，与指令处理两个二进制位(bit)，根据这两个位的计算结果产生第三个位。根据习惯，我们认为二进制位 1 的值为真，二进制位 0 的值为假。其中的逻辑是一样的，我们仅仅是使用不同的符号来表示真和假而已。根据这一点，我们可以改写与运算的真值表，使之对于汇编语言的工作更有意义(见表 9-2)。

表 9-2　汇编语言中的与真值表

位 1	操作符	位 0	结果位
0	AND	0	0
0	AND	1	0
1	AND	0	0
1	AND	1	1

9.1.3　与指令

X86 指令集中的与指令体现了这个概念。与指令对两个位数相同的操作数执行与逻辑运算，并且一般情况下，用计算结果替换目标操作数。(注意，目标操作数通常情况下是指最接近助记符的那个操作数。)换句话说，思考一下这条指令：

```
and al,bl
```

当 CPU 对 AL 和 BL 中的一组 8 位二进制数执行八次按位与操作时，将会发生什么。AL 中的第 0 位与 BL 中的第 0 位相与，AL 中的第 1 位与 BL 中的第 1

位相与，以此类推。每个与操作产生一个结果位，当所有这 8 个与操作完成后，这些结果位被放到目标操作数里(这里为 AL)。这是机器指令中惯用的一种思路，即：对两个操作数执行某种操作，用产生的结果替换第一个操作数(目标操作数)，而不是第二个操作数！

9.1.4 位屏蔽

与指令的一个主要应用是把一个二进制位或者是更多的二进制位从字节数据或者是字数据中分离出来。这里所说的分离，只不过是把所有不需要的二进制位都置为一个可靠的值：0。例如，假设我们感兴趣的是：测试某个数值的第四位和第五位，看一下这些位到底是什么。要想实现这一目标，我们必须能够忽略其他位(0～3 位，6～7 位)，而唯一能够安全忽略这些位的方法，就是将这几位都置为 0。

AND 指令就能做到这一点。我们设置一个掩码，其中，我们想观察和测试的那些位被设置成 1，而我们希望忽略的那些位则被设置为 0。要想把除了第 4 位，第 5 位之外的其他位都屏蔽掉，我们必须建立这样的掩码：第 4 位，第 5 位为 1，其他位都为 0。用二进制来表示，该掩码就是：00110000B 或者 30H。(为了证实这一点，可以从第 0 位开始，对这个二进制数从右往左进行计数。)然后将这个掩码和我们正在讨论的数值相与即可。图 9-2 展示了这个运算的操作过程，初始值为 9DH，掩码为上文所述的 30H。

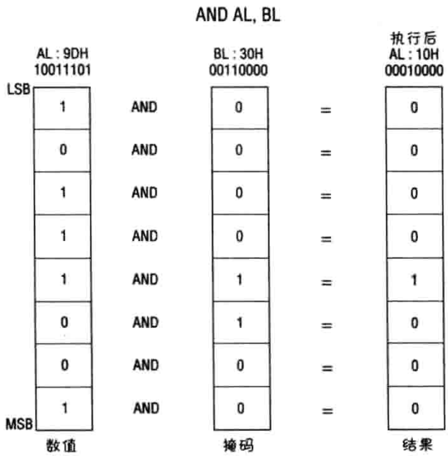


图 9-2　与指令剖析

涉及的 3 个二进制数被纵向排列，每个值的最不重要的位(即最右侧的那个位)在最上边。你应该能够跟踪每个与操作，并且能够通过参照表 9-2 来进行验证。最终结果是除了第 4 位和第 5 位之外的其他位都保证为 0，因而能够被安全地忽略。第 4 位和第 5 位可以是 0，也可以是 1。(这就是为什么我们要测试它们的原因，我们不知道它们的值是什么。)对于初始值 9DH，第 4 位的结果是 1，第 5 位的结果是 0。如果初始值是其他数，第 4 和第 5 位结果是可能都是 0，或者都为 1，或者为一些 0、1 的组合。

千万不要忘记：当操作完成时，与指令的运算结果替换的是目标操作数。

9.1.5 或指令

跟与逻辑操作关系最为密切的是或操作(OR)，类似于与逻辑操作，在 x86 指令集中，或操作的含义与其名字相同。在结构上，或指令跟与指令工作原理相同。只是真值表不同：要想让结果为 1，与操作要求两个操作数必须都为 1，而或操作只要求至少有一个为 1 即可。或操作的真值表如表 9-3 所示。

表 9-3　汇编语言的或真值表

位 1	操作符	位 0	结果位
0	OR	0	0
0	OR	1	1
1	OR	0	1
1	OR	1	1

由于或操作不适合做位屏蔽，因此比与运算用的少得多。

9.1.6 异或指令

异或操作是排他性的或指令，具体体现在 XOR 指令中。再次强调，像与操作和或操作一样，异或操作也是对它的两个操作数执行逻辑运算，并且将运算结果替换目标操作数。然而，这个逻辑操作是排他性的或，意思是：只有当这两个操作数的值不同(即 1 和 0 或者 0 和 1)时，结果才为 1。使用如表 9-4 所示的异或真值表有助于更清晰地展示异或的概念。

表 9-4　汇编语言的异或真值表

位 1	操作符	位 0	结果位
0	XOR	0	0
0	XOR	1	1
1	XOR	0	1
1	XOR	1	0

请仔细观察表 9-4！在第一种和最后一种情况里，两个操作数的值是一样的，运算结果为 0。而在中间的两种情况里，两个操作数的值是不同的，结果为 1。

可以用异或做一些有趣的事情，但是大多数会令初学者感到有些神秘。异或的一个不太明显的应用是：任何值与本身相异或，将产生结果 0。换句话说，如果你把寄存器作为操作数，对其进行异或操作的话，这个寄存器将会被清零，即被设置成 0:

```
xor eax,eax  ; 清零 eax 寄存器
```

在过去，这种方式比通过使用 MOV 来加载立即数 0 到寄存器要快一些。尽管现在已经不再这样，但是它仍然是一个有必要知道的有趣技巧。异或的工作原理已经非常清晰地展示在真值表中，但是要想完全把它讲解清楚，我打算采用图 9-3 来进行阐释。

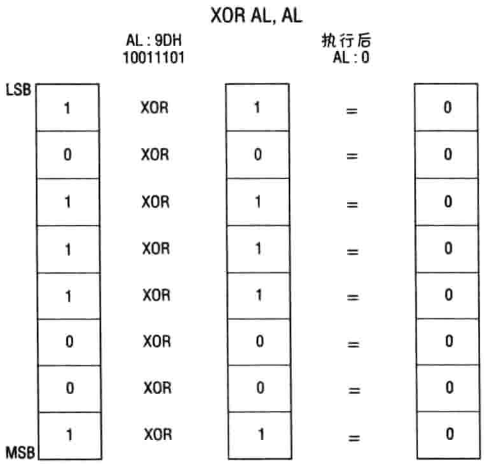


图 9-3　使用 XOR 清零寄存器

我们看一下图中每一个单独的异或操作及结果值。因为 AL 中的每一位都与自身相异或，在每次异或操作中都对两个相等的操作数进行操作。有时操作数都为 1，有时操作数都为 0，但是每种情况下这两个操作数都相同。在异或操作中，当两个操作数的值相同时，其结果总是为 0. 瞧，就这样，寄存器中被放入了数据 0。

9.1.7 非指令

在所有逻辑指令中，最容易理解的就是非指令。非操作的真值表也比我们前面看到的要简单，因为非操作仅仅对一个操作数进行运算。该运算做的事情也非常简单：非操作取出它的单操作数的每一位的状态，然后将其变为相反的状态。1 变成 0，0 变成 1，如表 9-5 所示。

表 9-5 汇编语言的非真值表

位	操作符	结果位
0	NOT	1
1	NOT	0

9.1.8 段寄存器对逻辑操作没有反应

你不能直接访问 X86 的段计存器，除非已经达到了可以对操作系统编程的水平。段寄存器属于操作系统，用户空间程序不能以任何方式修改它们。

但是，即使你已经开始在操作系统级别上编程了，段寄存器仍然具有很大的局限性。其中之一就是它们不能进行按位逻辑运算。如果企图这样做的话，汇编编译器会抛给你“非法使用段寄存器”的错误提示。如果需要对段寄存器执行逻辑操作的话，你必须首先将该段寄存器中的值拷贝到以下寄存器之一：EAX、EBX、ECX、EDX、EBP、ESI 或者 EDI；对通用寄存器执行逻辑运算，然后再将通用寄存器中的计算结果拷回段寄存器中。

通用寄存器被称为“通用的”是有原因的，段寄存器在任何方面都不是通用的。它们的内存地址是专有的，如果你必须修改段寄存器的值，一般的方法是将其挪到通用寄存器中去完成，然后再将修改后的值拷回所处理的段寄存器中。

9.2 移位操作

另外一种操纵字节中的位的方式要更直接一些：将这些位往某个方向(左或者右)移动。这需要处理很多细节，但是最简单的移位指令相当显而易见：SHL 将它的操作数左移，而 SHR 将它的操作数右移。

所有的移位指令(包括一些我将在稍后讲解的比较复杂的指令)都有相同的一般形式，如 SHL 指令所示：

```
shl <寄存器/内存>,<count>
```

第一个操作数是移位操作的目标——也就是那个即将进行移位操作的数值。它可以是寄存器数据或者内存数据，但是不能是立即数。第二个操作数则指出了即将移动的位数。

9.2.1 根据什么进行移位操作

表示移动位数的<count>操作数的由来有着独特的历史。在老式的 8086 和 8088 中，它可能是以下二者之一：立即数 1 或者寄存器 CL。(不是 CX!)如果你指定<count>为 1，那么移动位数就是 1 位，如果你希望一次移动不止一位，你必须把需要移动的位数装入寄存器 CL 中。在 X86 通用寄存器成为真正的“通用”之前，计数的工作成了 CX 寄存器的隐性议程。它通过循环、线性元素以及其他结构对移动的位数进行计数。这就是为什么它有时候被叫作计数寄存器，并且能够通过单词 Count 的首字母 C 来进行记忆的原因。

尽管能够移动的最大位数可达 255，但是真正有意义的移动位数最多到 32。如果想将一个双字中的某些位移动超过 32 位，你将会把它完全移出这个 32 位双字——更不用说字节或者字了！

从 286 以及所有更新的 X86 CPU 开始，<count>操作数可以是从 1 到 255 之间的任意立即数了。因为 Linux 需要至少运行在 386 上，所以当你在 Linux 下进行编程的时候，对于移位计数值的那些古老的限制已经不复存在了。

9.2.2 移位指令的工作原理

要想理解移位指令，需要把即将进行移位操作的数转换成二进制数，而不是

十六进制或者十进制数。(如果你对二进制数制系统感到模糊的话，回去重温第二章的内容。)让我们从一个存放数据 0B76FH 的寄存器 AX 开始一个简单的示例。将 0B76FH 表示成如下二进制数(因此以位的形式出现)：

```
1011011101101111
```

请时刻记住，二进制数中的每一个数字(digit)都是一个位(bit)。如果执行指令 SHL AX,1，你将发现，移位操作之后，AX 中的数据变成如下内容：

```
0110111011011110
```

在这个数的右端被插入了一个 0，整个数据都往左边移动了一位。注意，最左侧的二进制位 1 已经被从这个数中移除到“空白宇宙”之中，不复存在了。

9.2.3　将位移入进位标志

其实，确切来讲，也不是“空白宇宙”，从这个二进制数的左侧移出来的最后一位，被存放到名为进位标志的位临时存储段里，它一般缩写为“CF”　。进位标志是那些聚集在一起形成 EFlag 寄存器的信息位之一，EFlag 寄存器已经在第 7 章中进行了讲解。你可以使用分支指令来测试标志位的状态，我将在本章稍后的部分对其进行讲解。

然而，当使用移位指令时，请记住，许多不同的指令都使用进位标志(Carry flag)——不仅仅是移位指令。如果你把一个二进制位移入进位标志，目的是想测试一下这个位，然后看一下它的值到底是什么，一定要在执行其他能够影响进位标志的指令之前进行。这包括所有的算术指令、所有的按位逻辑操作指令以及其他一些不同种类的指令——当然，还有所有的其他移位指令。

如果你将一位移入进位标志，然后立即执行其他的移位指令，那么，第一个移入的位将被从这个数中移除，进入“空白宇宙“之中。

9.2.4　循环移位指令

如果希望一个二进制位的命运不是即将被丢弃到“空白宇宙”，需要使用循环移位指令：RCL、RCR、ROL 和 ROR。循环移位指令几乎和移位指令相同，但是有一个重要区别：从操作数的一端移出的二进制位，将会重新出现在操作数的另一端。当你对一个操作数执行不止一位的循环移位操作时，这些位不断地在

某一方向上进行匹配，从某一端被移除，然后立刻出现到另外一端。当循环指令被执行时，这些位就这样在操作数中循环。

采用图表显示会更直观些，请参见图 9-4。这里的例子是 ROL(循环左移)指令，循环右移指令工作原理相同，只不过位的移动方向与前者正好相反。AL 的初始值为二进制数 10110010 (0B2h)，当执行 ROL AL,1 指令时，所有 AL 中的二进制位都整齐地向左移动一位。第 7 位的二进制位 1 退出 AL 的左侧，但是转了一圈，立即重新出现在 AL 的右侧。

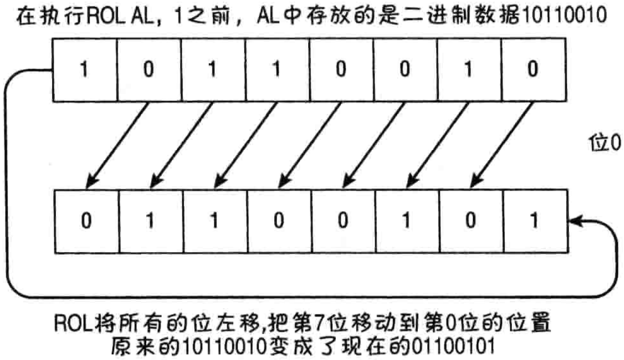


图 9-4 循环指令工作原理

再强调一下，ROR(循环右移)指令工作原理相同，只不过位的移动方向是从左往右，而不是 ROL 指令中的从右往左。操作数即将循环移动的位数可以是立即数，也可以是 CL 寄存器中的值。在 X86 指令集中，还有一对儿循环指令：RCR(带进位的循环右移指令)和 RCL(带进位的循环左移指令）。这些操作跟 ROL 和 ROR 类似，但是有一点不同：从操作数的某一端移出的二进制位，会通过进位标志来重新进入该操作数。任意一个位通过 CF 进行循环的路径，都比通过 ROL 和 ROR 进行循环的路径长一位。我已经以图形的方式将其展示在图 9-5 中。

如果对一个值循环移动超过 31 位，循环指令与移位指令相比并没有优势。(如果你把一个数循环移动 32 位，你将得到一个和开始循环的那个数一样的值。)循环移位指令将位从操作数的一端移出，然后再将它们放回到另一端，如此往复。当你在脑子里跟踪某个位的循环移位过程时，将会发现，经过 32 次循环之后，任何一个位都回到了循环之前的原来位置。一个二进制位如此，所有的二进制位亦如此，所以对于一个 32 位数而言，31 次就是有意义的最大的循环移位次数了。这就是为什么在保护模式下(以及在老式的 286 下)编程时，在指令执行之前，移位的数量被删减到 5 位的原因。归根到底，5 位二进制数所能表示的最大

值是 32！

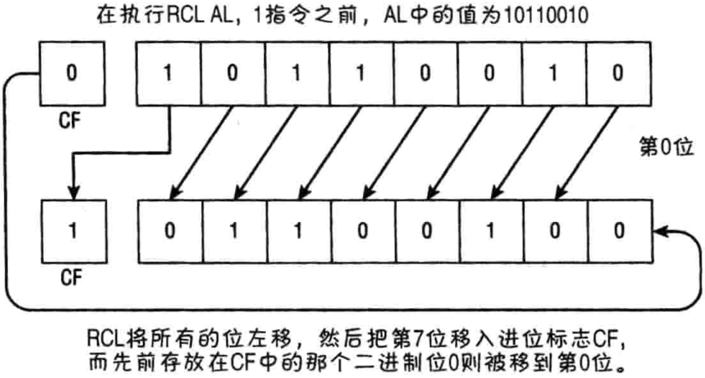


图 9-5 通过进位标志进行循环移位的指令的工作原理

9.2.5 将已知值存入进位标志 CF

记住这一点也很有用：前面的指令可能在 CF 中留下值，而这些值将会通过 RCL 或 RCR 指令，循环到一个操作数中。有些人错误地认为，在执行一个移位或者循环移位指令之前，CF 会被强制清零，事实却并非如此。如果另一条指令在 RCR 或 RCL 指令执行之前刚刚留下了一个二进制位 1 在 CF 中，那个 1 将会顺理成章地进入目标操作数，不管你想不想要它。如果循环开始之前在 CF 中存入一个已知值这一点非常重要，那么两条 X86 指令将会帮助你完成这个任务：CLC 和 STC。CLC 把标志位清零，STC 将标志位设置为 1。这两个指令都不需要操作数，除了能够改变 CF 中的值之外，它们都别无他用。

9.3 位操作

通过前几章的学习，我们知道，Linux 能够相当方便地在显示器上显示文本。问题是它只能显示文本——如果你想把某个寄存器中的值以十六进制的方式进行显示，Linux 就无能为力了。你必选先把这个数转换成字符串形式的表达方式，然后再通过 INT80h 调用 sys_write 内核服务来显示这个字符串。

把十六进制的数值转换成十六进制数元(digit)并不难，完成这项工作的代码体现了我们将在本章探索的若干新概念。清单 9-1 中的代码是十六进制转储工具中最为核心的部分，颇像一个 Bless 十六进制编辑器的只读版本。当你从任意类

型的文件中重定向它的输入时，它都会从该文件中一次读入 16 个字节，并将其显示在一行里，形成由空格隔开的 16 进制值。

清单 9-1：hexdump1.asm

```
; 可执行(文件)名 : hexdump1
; 版本 : 1.0
; 创建日期 : 2009/4/4
; 最后更新日期 : 2009/4/4
; 作者 : Jeff Duntemann
; 描述 : Linux 下的一个简单的汇编程序，采用 NASM 2.05,
;        演示了如何把二进制值转换成 16 进制字符串表达。
;        本程序可作为供文件使用的一个非常简单的 16 进制转换工具，
;        尽管没有 ASCII 等值列
; 通过以下方式执行本程序:
; hexdump1 < (input file)
;
; 通过以下命令生成本程序:
; nasm -f elf -g -F stabs hexdump1.asm
; ld -o hexdump1 hexdump1.o
;
SECTION .bss                    ; 包含未初始化数据的段

     BUFFLEN equ 16             ; 我们一次从这个文件读取 16 个字节
     Buff: resb BUFFLEN         ; 文本缓冲区本身

SECTION .data                   ; 包含已初始化数据的段
     HexStr: db " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00",10
     HEXLEN equ $-HexStr

     Digits: db "0123456789ABCDEF"

SECTION .text                   ; 包含代码的段

global _start                   ; 连接器需要这个来找到入口点!

_start:
     nop                        ; 这个无操作指令使得 gdb 感到非常高兴…

; 从标准输入中读取满满一缓冲区的文本:
Read:
     mov eax,3                  ; 指定 sys_read 调用
     mov ebx,0                  ; 指定文件描述符 0: 标准输入
```

```
    mov ecx,Buff              ; 传递将要从中读入数据的缓冲区的偏移地址
    mov edx,BUFFLEN           ; 传递一次需要读取的字节数
    int 80h                   ; 调用 sys_read 来填充缓冲区,
    mov ebp,eax               ; 存储从文件中读入的字节数以备后用
    cmp eax,0                 ; 如果 eax=0,则 sys_read 已经到达标准输入结尾
    je Done                   ; (通过与零比较) 相等则跳转

; 设置寄存器用于处理缓冲区步骤:
    mov esi,Buff              ; 将文件缓冲区的地址存入 esi
    mov edi,HexStr            ; 将线性字符串的地址存入 edi
    xor ecx,ecx               ; 将线性字符串指针清 0

; 通读检查缓冲区，将二进制数值转换为 16 进制位元(hex digits):
Scan:
    xor eax,eax               ; 将寄存器 eax 清零

; 这里我们可以计算相对于 HexStr 的偏移地址，它等于 ecx 中的值乘以 3
    mov edx,ecx               ; 将字符计数器的值复制到 edx 中
    shl edx,1                 ; 通过左移位操作将指针的值乘以 2
    add edx,ecx               ; 完成乘以 3 操作

; 从缓冲区中获取一个字符，并将其同时存入 eax 和 ebx 中:
    mov al,byte [esi+ecx]     ; 从输入缓冲区中取出一个字节，送入 al
    mov ebx,eax               ; 为了求得第二个“半字节”，将该字节复制到 bl 中

; 查找低半字节字符并将其插入字符串:
    and al,0Fh                    ; 屏蔽除了低半字节之外的所有内容
    mov al,byte [Digits+eax]      ; 查找与该半字节相等的字符
    mov byte [HexStr+edx+2],al    ; 将最低有效位(LSB)字符位元(digit)写
                                  入线性字符串

; 查找高半字节字符并将其插入字符串:
    shr bl,4                      ; 将字符的高四位移入低四位
    mov bl,byte [Digits+ebx]      ; 查找与该半字节相等的字符
    mov byte [HexStr+edx+1],bl    ; 将最高有效位(MSB)的字符位元写入线性字符串

; 将缓冲区指针指向下一个字符，看一下任务是否完成:
    inc ecx                       ; 增加线性字符串指针的值
    cmp ecx,ebp                   ; 与缓冲区中的字符数进行比较
    jna Scan                      ; 如果 ecx<=缓冲区中的字符数，转回去继续循环

; 将这行十六进制值写入标准输入:
```

```
        mov eax,4                          ; 指定 sys_write 调用
        mov ebx,1                          ; 指定文件描述符 1：标准输出
        mov ecx,HexStr                     ; 传递线性字符串的偏移地址
        mov edx,HEXLEN                     ; 传递线性字符串的大小
        int 80h                            ; 通过内核调用来显示线性字符串
        jmp Read                           ; 继续循环，再次加载文件缓冲区

; 胜利完成任务！让我们来结束这场“派对”：
Done:
        mov eax,1                          ; 该行代码用于退出系统调用(Syscall)
mov ebx,0                                  ; 将返回值设为 0
        int 80H                            ; 进行内核调用
```

hexdump1 本质上是一个过滤器程序，与第 8 章中的 uppercaser2 程序具有同样的通用过滤器机制。下面就该程序的重要部分展开讨论，即从输入缓冲区中读入 16 个字节，并将其转换成一个字符串，显示在 Linux 控制台上。这些是位于标号 Scan 和 INT 80h 退出调用之间的代码。在本章后面部分的讨论中，还要用到这个代码块。

9.3.1　将一个字节分解为两个“半字节”

记住，Linux 以二进制的形式将值从文件中读入内存。十六进制是一种显示二进制值的方式，为了能以可以显示的十六进制的 ASCII 码来显示二进制值，我们必须进行一些转换操作。

显示一个 8 位二进制数需要两个十六进制数。字节的低四位(最不重要的，或者最右侧的位)用一个十六进制数来表示，高四位(最重要的，或者最左侧的位)用另外一个十六进制数表示。例如，二进制数 11100110 与十六进制数 E6 相等。(我在第二章中已经对此进行了详细介绍。)把一个 8 位二进制数转换成两个 4 位二进制数时，必须每次转换 1 个，这意味着我们不得不把一个单字节拆分成 2 个 4 位大小的二进制量，我们通常将其称作半字节(nybble)。

在 hexdump1 中，我们将一个字节从缓冲区中读出，然后放入 EAX 和 EBX 两个寄存器中。之所以这样做，是因为把一个字节的高位和低位分开具有一定的破坏性，因为从本质上来讲我们把不需要的那个半字节清零了。

要从字节中分离出低 4 位，我们需要屏蔽掉不想要的那个半字节。这可以通过与指令来完成：

```
and al,0Fh
```

立即数常数 0Fh 用二进制表示为 00001111。如果通过与真值表(表 9-2)来跟踪这个操作，你会发现，任意二进制位与 0 相与，结果都为 0。我们把寄存器 AL 中的高半字节和 0000 相与，就可将所有可能出现在该半字节的内容全部清零。将低半字节和 1111 相与则会把低字节毫发无损地精确保留。

当我们完成了这些，也就成功地把这个字节的低半字节从缓冲区中读入了 AL 里。

9.3.2　将高半字节移入低半字节

屏蔽掉 AL 中的输入字节的高半字节将会破坏它的值。我们需要高半字节，但是由于在 EBX 中保留了输入字节的另一份拷贝，我们可以通过它来导出高位字节。至于低半字节，我们实际上是在处理 EBX 的最不重要的那 8 位，将之作为 BL 中的值。记住，BL 仅仅是引用 EBX 低 8 位的另外一种方式。如果一个数被加载到 EBX 中，它的最不重要的 8 位被存放在 BL 中。

我们可以通过与指令屏蔽掉 BL 中的低半字节，只留下高半字节，但是这里有一个圈套：屏蔽掉一个字节的低 4 位并不会把高 4 位变成一个“半字节”。我们必须以某种方式把输入字节的高 4 位移到低 4 位。

此时，我们已经把输入字节的低半字节存入了 AL 中，而高半字节则存入了 BL 中。下一个挑战就是把“半字节”中 4 位二进制数(如 1110) 转换成对应的十六进制位元的 ASCII 字符——本例中，是字符 E。

9.3.3　使用查找表

在程序的.data 部分，定义了一个非常简单的查找表。Digits 表定义如下：

```
Digits db '0123456789ABCDEF'
```

关于 Digits 表，需要注意的重要事情是：每一个十六进制位元(digit)都在这个字符串中占用一个位置，而该位元相对于字符串起始位置的偏移地址就是它所代表的值。换句话说，ASCII 字符“0”位于这个字符串的最开始位置，它相对于这个字符串的起始位置的偏移地址是 0 个字节。字符“7”位于字符串起始位置的第 7 个字节，以此类推。

我们可以通过使用一个内存引用来从 Digits 表中“查找”一个字符：

```
mov al,byte [Digits+eax]
```

和大多数汇编语言一样，这里的所有一切都取决于内存寻址。查找表中的第一个十六进制位元字符位于 Digits 所包含的地址处。要想获得所需要的位元(digit)，我们必须为查找表建立索引。这可通过把查找表地址和相对于该表的起始位置的偏移地址相加放入方括号内来实现。

把 AL 中存放的偏移地址和 Digits 查找表的地址(通过 EAX)相加刚好能够把与 AL 中的值相等的 ASCII 值字符带给我们。我已经将其实现过程以图形的方式展示在图 9-6 中。

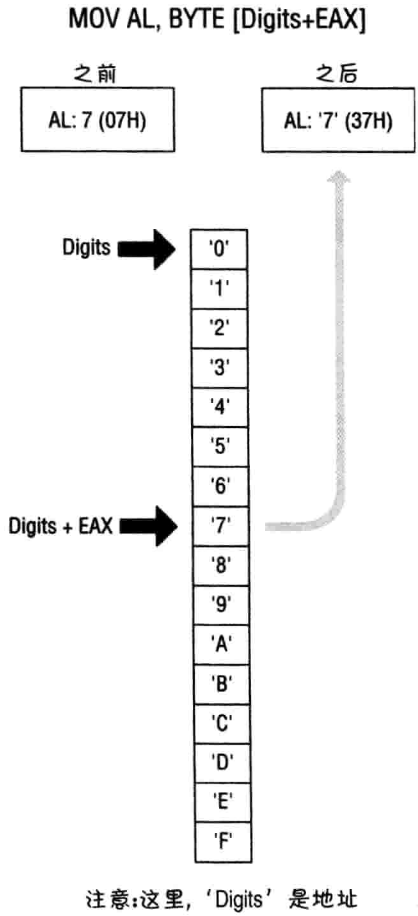


图 9-6 使用查找表

关于从查找表 Digits 中取出一个位元并将其放入 AL 中的 MOV 指令，可能有两件令人费解的事情：

- 在内存引用中，我们必须使用 EAX 而不是 AL，因为 AL 不能参与有效地址计算。不要忘记，AL 位于 EAX“内部”！(关于有效地址计算的更多内容，将在本章稍后给出。)
- 我们是把 AL 中的半字节用与之相等的字符进行替换。该指令首先从查找表中取出与这个半字节相等的字符，然后再将这个相等的字符存回 AL。而之前 AL 中的那个半字节则不复存在了。

到目前为止，我们已经从查找表中读入了一个字符到 AL 中。该半字节的转换已经完成。下一个任务听起来比较简单，但是实际上则非常棘手：把现在 AL 中的那个十六进制 ASCII 码写入显示字符串 HexStr。

9.3.4　通过移位和相加来实现相乘

hexdump1 程序从一个文件中读出字节，然后将其按行显示，每行显示 16 个以十六进制表示的字节。程序的部分输出如下：

```
3B 20 20 45 78 65 63 75 74 61 62 6C 65 20 6E 61
6D 65 20 3A 20 45 40 54 53 59 53 43 40 4C 4C 0D
0A 3B 20 20 56 65 72 73 69 6F 6E 20 20 20 20 20
20 20 20 20 3A 20 30 2E 30 0D 0A 3B 20 20 43 72
65 60 74 65 64 20 64 60 74 65 20 20 20 20 3A 20
30 2F 37 2F 32 30 30 39 0D 0A 3B 20 20 4C 60 73
74 20 75 70 64 60 74 65 20 20 20 20 20 3A 20 32
2F 30 38 2F 32 30 30 39 0D 0A 3B 20 20 40 75 74
68 6F 72 20 20 20 20 20 20 20 20 20 20 3A 20 4A
```

所有这些行都在显示同一项内容：HexStr，一个由 48 个字符组成的、带有 EOL 结尾标志的字符串。当 hexdump1 每次从输入文件中读入一个 16 字节的数据块时，它都将其转换成十六进制位元的 ASCII 字符格式，并插入 HexStr 中。从某种意义上来讲，这是另外一种类型的表操作，只不过我们将查表操作替换为根据索引将一些值写入表中而已。

另外一种方式是将 HexStr 看成一个具有 16 个表项的表，每一个表项长度为 3 个字符(参照表 9-7)。在每一个表项中，第一个字符是空格，第二个和第三个字符是该 16 进制数本身。在.data 段中，作为 HexStr 的初始化定义的一部分，空格字符已经存在了。初始的“空”HexStr 字符串在所有 16 进制位元的位置都有一

个 0 字符，为了给 HexStr 填充真实的数据，以便显示每一行，我们必须通过汇编循环来扫描 HexStr，将低半字节字符和高半字节字符分开来写。

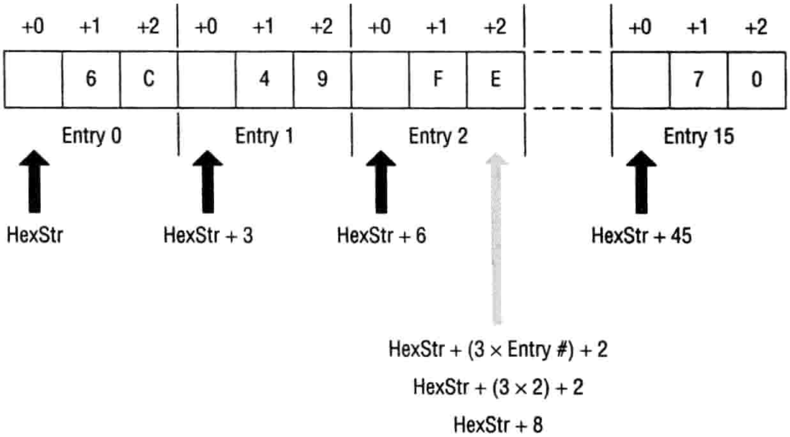


图 9-7　具有 16 个 3 字节表项的表

这里棘手的事情是：对于每一次循环，我们都不得不“猛击”HexStr 索引 3 次而非 1 次。HexStr 中的那些 3 字节条目的偏移地址是该条目索引的 3 倍。我已经将讲解了 MUL 指令，它能处理 X86 指令集中任意的无符号乘法。该指令同样具有一些局限性，尤其是它需要特定的寄存器来完成隐式操作的工作方式。

所幸的是，在汇编语言中，有一些其他的、速度更快的方法可实现乘法操作，只要你的脑袋稍微灵活一点点。这些方法基于这一事实：将一个数与 2 的幂数相乘可以非常容易和快捷地实现，使用 SHL(左移位)指令即可。也许对你来说，这一点不是很显而易见，但是每次你将一个数实现左移位一位时，实际上你是在将这个数与 2 相乘。将一个数实现左移位两位时，实际上你是在将这个数与 4 相乘。将一个数实现左移位 3 位时，实际上你是在将这个数与 8 相乘，以此类推。

你可以通过我的描述来进行理解，也可以“实地考察”一下到底“沙盒”里发生了什么。建立一个新的沙盒程序，将如下指令放入其中：

```
mov al,3
shl al,1
shl al,1
shl al,2
```

生成这个沙盒程序并将可执行程序加载到 Insight 中。在寄存器视图中将

EAX 显示域设置为十进制视图。(必须在这个沙盒程序已经运行后方可这样做，通过右击 EAX 域并从弹出的上下文菜单中选择十进制来实现。)然后单步执行这些指令，在寄存器视图下观察每一步 EAX 中值的变化情况。

第一条指令将数值 3 加载到 AL 中。下一条指令将 AL 中的数据实现左移一位操作。AL 中的值变成了 6。第二条 SHL 指令又把 AL 中的数据左移了 1 位，AL 中的数由 6 变成了 12。第三条 SHL 指令把 AL 中的数据左移了 2 位，AL 中的数由 12 变成了 48。我将这些内容以图像化的方式展示在图 9-8 里。

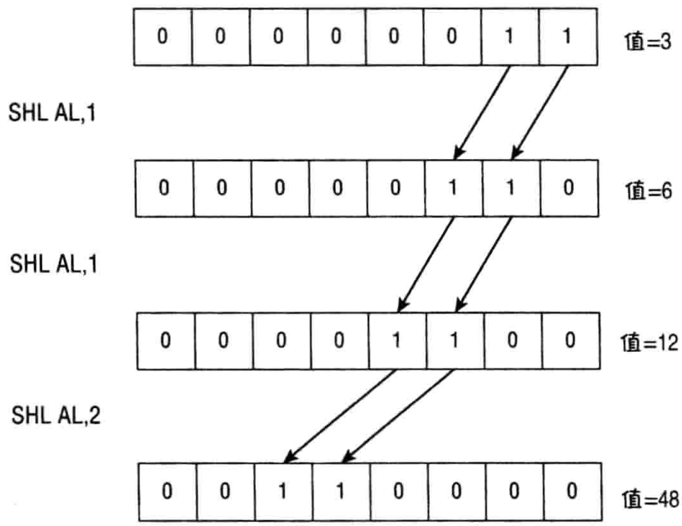


图 9-8 通过移位实现相乘

如果想实现将一个数与 3 相乘，该怎么办呢？非常容易：你先将这个数与 2 相乘，然后再加把得到的结果与被乘数的一份复制值相加即可。在 hexdump1 程序中，该操作是这样完成的：

```
mov edx,ecx         ; 将字符计数器复制到 edx
shl edx,1           ; 通过左移位将指针与 2 相乘
add edx,ecx         ; 完成乘以 3 操作
```

这里，我们将被乘数从循环计数器 ECX 中加载到 EDX 中。然后将 EDX 左移一位来实现与 2 相乘。最后，将 ECX 再次与 EDX 中的结果相加，进而实现将其与 3 相乘的效果。

与非 2 的幂数相乘可以通过将一条 SHL 指令与一条或多条加法(ADD)指令组

合来实现。例如，想把一个 ECX 中的值与 7 相乘，可以这样做：

```
mov edx,ecx          ; 将字符计数器复制到 edx
shl edx,2            ; 将 edx 与 4 相乘
add edx,ecx          ; 实现 edx 与 5 相乘
add edx,ecx          ; 实现 edx 与 6 相乘
add edx,ecx          ; 实现 edx 与 7 相乘
```

这样做看起来似乎很笨拙，但是即使这样它依然比 MUL 运算速度快！(在本章稍后的部分，我还会向你展示一种甚至更快的方法来实现与 3 相乘。)

一旦理解了字符串表 HexStr 如何创建，就可以直截了当地把十六进制位元系列写入它。最低有效十六进制位元在 AL 中，最高有效十六进制位元在 BL 中。通过一个由 3 部分组成的内存地址来实现将这两个十六进制位元写入十六进制字符串(HexString)：

```
mov byte [HexStr+edx+2],al ; 将 LSB 字符位元(char digit)位写入字符串行
mov byte [HexStr+edx+1],bl ; 将 MSB 字符位元(char digit)位写入字符串行
```

参考前面的图 9-7，独立完成如下内容：从 HexStr 的地址开始，从头至尾做一遍。EDX 包含了某一指定表项的第一个字符的偏移地址。为了获取该表项的地址，需要把 HexStr 和 EDX 相加。然而，这个地址通常是该表项的第一个字符的地址，而该字符通常在 HexStr 中是一个空格字符。每一个表项的 LSB 位元地址为该表项偏移地址+2，MSB 位元地址为该表项偏移地址+1。因此，LSB 位元的地址是 HexStr+表项的偏移地址+2。MSB 位元的地址是 HexStr+表项的偏移地址+1。

9.4　标志、测试和分支

从某一高度来看，条件转移指令的思想很简单，而且如果没有它，很多汇编操作将无法完成。在接下来的几个例子中，我会经常非正式地应用条件转移指令，但是不对其作过多解释，因为根据上下文可以相当明显地看出转移(jump)的含义，这些转移指令是展示一些其他内容的必不可少的组成部分。但是，在看似简单的汇编语言转移指令思想的背后，却蕴涵着很多错综复杂的内容。我想现在是时候对它们进行完整详细的讲解了。

9.4.1　无条件转移

所谓转移，是指指令流执行过程中的一个生硬的改变。通常情况下，指令是一条一条地从低内存到高内存顺序执行的。条件转移改变了即将执行的下一条指令的地址。执行一个转移指令，嗖的一下，突然间你就到了别的地方。转移指令可以在内存中移动执行的位置。它可以使执行路径折回去构成一个循环(它还能将程序绑定到诸多逻辑关系之中)。

有两种类型的转移指令：条件转移指令和无条件转移指令。无条件转移指令是指肯定会发生的跳转。它的格式如下：

```
jmp <label>
```

当这条指令执行时，执行顺序转到了位于标号<label>处的那条指令。就这么简单。

9.4.2　条件转移指令

条件转移指令是我在第一章中介绍的那些虚构的测验之一。当执行时，条件转移指令需要检验一些东西，通常是一个，有时候两个，或者很少情况下，3 个 EFlags 寄存器中的标志位。如果被检验的标志位或者标志位们碰巧处于某一特定状态，执行将会跳转到代码段中其他位置的某个标号处；否则，它执行失败，继续按顺序执行下一条指令。

这种两面性非常重要。一个条件转移指令要么跳转成功，要么跳转失败。要么跳转，要么不跳转。不可能跳转到两个位置之一，更不能跳转到三个位置之一。是否执行跳转取决于 CPU 内部的一小撮二进制位序列的当前值。

在本书的前面部分，当从整体上讨论 EFlags 寄存器时，我曾经提到，当某条指令的执行结果为 0 时，会有一个标志位的值被某些特定指令设置为 1：零标志(ZF)。DEC(递减)指令就是一个很好的例子。DEC 把它的操作数减 1。如果通过减操作使得这个操作数变为 0，ZF 就被设置为 1。其中一个条件转移指令，JZ(为零跳转)，会检测 ZF 的值。如果发现 ZF 为 1，则发生跳转(转移)操作，执行转到 ZF 助记符后面的标号所指定的位置。如果发现 ZF 为 0，跳转失败，程序会继续往下按顺序执行下一条指令。它也许是整个 X86 指令集中最常见的条件转移指令了。当执行某个循环的时候，我们经常使用该指令来判断存放在寄存器中的计数

值是否为零。当计数器寄存器中的值通过 DEC 指令变为零时，这个循环结束，程序从循环之后的语句开始执行。

这里有一个简单的例子，其中使用的指令你应该已经能够全部理解 (尽管有些怪诞)：

```
mov word [RunningSum],0   ; 清零 [RunningSum]
mov ecx,17                ; 我们将执行此操作 17 次
WorkLoop:
add word [RunningSum],3   ; 将 [RunningSum]与 3 相加
dec ecx                   ; 循环计数器减 1
jz SomewhereElse          ; 如果计数器的值为零，任务完成!
jmp WorkLoop
```

在循环开始之前，我们将一个值存入 ECX 中，这里 ECX 作为计数器寄存器来使用，里面包含了即将执行的循环次数。循环体就是每一次循环操作执行内容。在本例中，循环体很简单，只有一条 ADD 指令，但是，它可以长达几十条甚至几百条指令。

当循环任务完成，计数器寄存器中的值被 DEC 指令减 1。紧随其后，JZ 指令检测零标志位。ECX 从 17 减到 16 或者从 4 减到 3 都不会设置 ZF，JZ 指令仅仅是执行失败。JZ 指令后面的指令是一条无条件转移指令，它每次都坚持将执行跳转到标号 WorkLoop 的位置。

现在，ECX 从 1 递减为 0，这才真正将 ZF 设置为 1，此时循环结束。最后，JZ 指令通过跳转到其他某处(它是一个标号，位于此处没有显示的更大的程序范围中)将我们带出循环，然后执行离开循环。

如果你很敏锐的话，会发现这种建立循环的方法非常差劲。(这并不意味着它不能完成循环，或者你自己在午夜极度不耐烦的情况下不会采用这种方式。)我们需要在每次循环中真正查找的是什么时候条件(即零标志位)没有被设置，以及那条完成该设置的指令。

9.4.3　条件“缺席”时进行跳转

有很多种条件转移指令，我会对其中一些进行讲解，但是本书不会全盘讲解。条件转移指令的数量在增加，因为每一个条件转移指令都有一个“密友”：当指定条件的值没有被设置为 1 时跳转。

JZ 指令是条件转移的很好示例。如果零标志位(ZF)被设置为 1，则 JZ 跳转

到代码段中的一个新的位置。JZ 的“密友”是 JNZ(不为零跳转)。当 ZF=0 时，JNZ 跳转到某个标号处，ZF=1 时，则跳转失败。

乍一听也许比较令人费解，因为当 ZF=0 时，JNZ 执行跳转。记住，指令的名字适用于被测试的条件，不一定适用于标志的二进制位值。在前面的示例代码中，当 DEC 指令将计数器的值减少为 0 时，JZ 跳转。被测试的条件是某些与前面某条指令相关的内容，不仅仅是 ZF 的状态。

按照如下方式进行思考：条件产生了标志。“产生了标志”意思是将标志设置位 1。当 n 条指令中的某一个将一个操作数的值变为零时(条件就是它)，零标志就产生了。该指令的逻辑指的是条件，而不是标志。

作为例子，让我们把前面的那个小循环改进一下(通过使用 JNZ 来改变循环逻辑)：

```
mov word [RunningSum],0   ; 清零 [RunningSum]
mov ecx,17                ; 我们将执行此操作 17 次
WorkLoop:
add word [RunningSum],3   ; 将 [RunningSum]与 3 相加
dec ecx                   ; 循环计数器减 1
jnz WorkLoop              ; 如果计数器的值为零，任务完成！
```

JZ 指令已经被代换为了 JNZ，这更加容易让人理解。因为为了使循环闭合，我们必须跳转，并且当计数器的值大于 0 的时候我们才关闭循环。只有计数器的的值大于零时，才会发生跳回标号 WorkLoop 的操作。

一旦计数器减小到 0，我们就认为循环完成了。JNZ 执行失败，而该循环后面的那条代码(这里没有写)将得到执行。要点是，如果能够立即定位程序在 JNZ 指令之后将执行的下一个任务，我们根本不需要使用 JMP 指令。指令执行很自然地流入下一个需要执行的任务。程序将具有一个更为自然的、不太费解的自顶向下流程，也就更加容易阅读和理解了。

9.4.4　标志

在前面的第 7 章中，我介绍了 Eflags 寄存器并且简单描述了它包含的所有标志位的目标。大部分标志不是很有用，尤其是当你刚刚步入程序员生涯的时候。进位标志(CF)以及零标志(ZF)将是新手用到的 90%的标志位。与方向标志(DF)、符号标志(SF)以及溢出标志(OF)一起，将会占到 99.90%。现在，为了防止对于与标志相关的知识已经生疏，建议重新阅读一下第 7 章中的相应部分。

如前所述，当 ZF=1 时，JZ 跳转，而当 ZF=0 时，JNZ 跳转。大多数对于某个操作数执行某种操作的指令(例如 AND、OR、XOR、INC、DEC 以及所有的算术指令)都会根据操作结果设置 ZF 的值。另一方面，那些仅仅执行数据移动的指令(例如 MOV、XCHG、PUSH 以及 POP)并不会影响 ZF 或者任何其他标志。(很明显，POPF 指令可通过将栈顶值弹出到标志寄存器中来影响它们。)一个很令人生气的例外是 NOT 指令，它对操作数执行一个逻辑操作，但是却并不设置任何标志——即使它会导致操作数的值变为零。在开始编写依赖于标志位的程序之前，最好查阅一下指令指南，以确保这些指令严格地正确遵守这些标志的规矩。如果不诡异的话，X86 指令集就不是 X86 指令集了。

9.4.5 通过 CMP 进行比较操作

标志位的一个主要应用是控制循环。另外一个主要应用就是对两个数进行比较。程序经常需要知道到底某个寄存器中或者内存中的值是不是与某些其他值相等。而且，你可能需要知道，如果与某个值不相等的话，到底是比那个值大还是比它小。有一条转移指令能够满足以上所有需求，但是，为了转移指令的执行，必须有某些东西来设置标志位。而 CMP(比较)就是那条为了完成比较任务而设置标志位的指令。CMP 指令的使用比较简洁和直观。它将第二个操作数与第一个操作数进行比较，然后根据结果设置相应的标志：

```
cmp <op1>,<op2>        ; 设置 OF, SF, ZF, AF, PF, 以及 CF
```

如果简单地以算数术语来“再造”比较操作的话，那么我想比较的含义就更容易牢记了。

```
Result = <op1> - <op2>
```

CMP 是一个减法操作，但是却丢弃减法的结果，只影响标志位。第二个操作数被从第一个操作数中减掉。根据相减的结果，相关标志位被设置成适当的值。

在 CMP 指令之后，你可以根据若干个算数条件进行跳转。那些具有很好的数学基础以及 FORTRAN 或 Pascal 基础的程序员，会更容易识别出以下条件：等于、不等于、大于、小于、大于等于、小于等于。这些操作符都见名知义，并且与大多数高级语言中的同名操作符非常类似。

9.4.6 转移指令的错综复杂之处

有一组令人费解的转移指令助记符，它们处理的数学关系一共分为六类，每一类对应着前面的六个条件之一。由于每一条机器指令都有两个助记符，这使得情况变得有些复杂——例如 JLE(如果小于等于则跳转)和 JNG(如果不大于则跳转)。这两个助记符是同义词，这是因为当遇到二者之一的时候，汇编语言产生相同的二进制操作码。此类同义词为程序员提供了方便，因为它们为考虑某一给定转移指令提供了两种可选途径。在前述示例中，如果小于等于则跳转逻辑上等于如果不大于则跳转。(请思考一下！)如果前面的比较的重要目的是判断一下某个值是否小于等于另一个值，可以使用 JLE 助记符。相反，如果测试的目的是判断某个值是否不大于另一个值，可以使用 JNG。选择权在你手中。

另外一个复杂之处是针对有符号算术比较和无符号算术比较都有单独的指令系列。本书已经对汇编语言数学做了很多介绍，这里就不再赘述有符号数和无符号数的区别了。有符号数的最高位被认为是一个内置的、指示该值是否为负数的标志。如果该位的值为 1，那么这个数被认为是负数。如果该位的值为 0，那么这个数被认为是正数。

汇编语言中的有符号算术运算是复杂且微妙的，而且不像你乍想之下那么有用。本书不会对其进行详细的讲解，尽管几乎所有的汇编语言书籍都对其从某种程度上进行了描述。为了对有符号算术理解得更深入一些，你需要知道的所有内容为：在有符号算术中，负数是合法的，而且一个值的最重要的位(即最高位)被当成符号位。(如果符号位为 1，则认为该数为一个负数。)

另一方面，无符号算术运算并不识别负数，最高有效位只不过是那个量所代表的二进制数中的一个普通位而已。

9.4.7 “大于”与“以上”

为了区分有符号转移指令和无符号转移指令，助记符使用两个不同的表达式来表示两个值之间的关系：

- 有符号数被视为大于或者小于。例如，为了测试一个有符号操作数是否大于另一个有符号数，需要在 CMP 指令后面使用 JG(如果大于则跳转)助记符。
- 无符号数被视为以上或者以下。例如，为了判断一个无符号操作数是否

大于(above)另外一个，需要在 CMP 指令后面使用 JA(如果以上则跳转)助记符令。

表 9-6 总结了算数转移助记符以及对应的同义词。所有包含以上或者以下的助记符都适用于无符号数，而所有包含大于或者小于的助记符则适用于有符号数。比较一下助记符和相应的同义词，注意它们是如何从不同的角度看待相同的指令的。

表 9-6 转移指令助记符及其同义词

助记符		同义词	
JA	如果以上则跳转	JNBE	如果不以下或者等于，则跳转
JAE	如果以上或者等于则跳转	JNB	如果不以下则跳转
JB	如果以下则跳转	JNAE	如果不以上或者相等，则跳转
JBE	如果以下或者等于则跳转	JNA	如果不以上，则跳转
JE	如果相等则跳转	JZ	如果结果为零则跳转
JNE	如果不等则跳转	JNZ	如果结果不为零则跳转
JG	如果大于则跳转	JNLE	如果不小于等于则跳转
JGE	如果大于等于则跳转	JNL	如果小于则跳转
JL	如果小于则跳转	JNGE	如果不大于等于，则跳转
JLE	如果小于等于则跳转	JNG	如果不大于，则跳转

表 9-6 将助记符扩展为更为易懂的形式，并且将每个助记符与其同义词进行了关联。另一方面，表 9-7 根据逻辑条件和它们在有符号数和无符号数中的使用方法，将助记符进行了分类。在表 9-7 中还列出了每一条转移指令测试的标志位。注意，为了执行跳转操作，某些转移指令可能需要测试两个可能的标志值之一，而有些则需要测试两个标志值。

表 9-7 CMP 指令执行后的有用算术测试

条件	PASCAL 操作符	无符号值	跳转条件	有符号数	跳转条件
等于	=	JE	ZF=1	JE	ZF=1
不等于	< >	JNE	ZF=0	JNE	ZF=0

续表

条件	PASCAL 操作符	无符号值	跳转条件	有符号数	跳转条件
大于	>	JA	CF=0 并且 ZF=0	JG	ZF=0 或者 SF=OF
小于	<	JB	CF=1	JL	SF< >OF
大于等于	> =	JAE	CF=0	JGE	SF=OF
不小于		JNB	CF=0	JNL	SF=OF
小于或者等于	< =	JBE	CF=1 或者 ZF=1	JLE	ZF=1 或者 SF< >OF
不大于		JNA	CF=1 或者 ZF=1	JNG	ZF=1 或者 SF< >OF

某些有符号转移指令需要比较两个标志位。例如，JG，当 ZF=0，或者 SF 与 OF 相等时，则执行跳转，这里我不会花费更多的时间来解释符号标志和溢出标志的本质含义。只需对这些指令有些初步了解即可，至于理解这些指令具体如何检验标志，可以等到我们已经具备了一些编程经验之后再来完成。

有些人对于 JE 和 JZ 助记符以及 JNE 和 JNZ 如何成为同义词的理解起来有些困难。我们再次想一下比较操作在 CPU 内部的完成方式：第二个操作数被从第一个操作数中减掉，如果结果为 0(实际上代表两个操作数相等)，那么零标志被设置位 1。这就是为什么 JE 和 JZ 是同义词的原因：二者都简单地检验零标志的状态。

9.4.8　使用 TEST 指令查找位 1

X86 指令系列意识到，汇编语言中经常用到位检测操作，于是，它提供了与 CMP 指令具有相同功能的指令：TEST。TEST 对两个操作数执行与逻辑操作，然后根据执行结果设置相应的标志位，但是并不修改目标操作数。TEST 指令语法如下：

```
test <操作数>,<位掩码>
```

位掩码操作数应该在操作数的每一个需要查找值是否为 1 的位置包含二进制位 1，而将其余位设置为 0。

TEST 指令所做的事情就是对该条指令的目标操作数和位掩码执行与逻辑操作，然后，根据与指令将会影响的标志位来进行标志位设置。与操作的结果将被丢弃，并且目标操作数不发生改变。例如，如果想判定 AX 的第 3 位是否为 1，可以使用如下指令：

```
test ax,08h           ; 用二进制表示的第 3 位的值为 00001000B, 或 08h
```

当然，第 3 位并不是指它包含数字 3——你必须观察掩码的位模式，然后将其用二进制或者十六进制表示。(用二进制表示的话，第 3 位代表数值 8。)使用二进制数作为字面常量在 NASM 中是完全合法的，而且通常是我们进行位掩码操作时的最清晰的表达方式。

```
test ax,00001000B     ; 用二进制表示的第 3 位的值为 00001000B, 或 08h
```

操作完成后，目标操作数 AX 中的值并没有改变，但是二进制模式 00001000 和 AX 却的的确确按照逻辑与真值表的规则执行了与操作。如果 AX 的第 3 位值为 1，则零标志被设置成 0。如果 AX 的第 3 位值为 0，则零标志被设置成 1。为什么呢？如果把(位掩码中的)1 和(AX 中的)0 相与，将得到 0。(可查找前述表 9-2 中与真值表。)如果所有的 8 个按位与操作结果都为 0，结果就是 0，零标志就会变为 1，用以指示结果为零。

理解 TEST 指令的关键是将其看成一种“操作码幽灵”，操作码是 AND。TEST 带上一个面具并且假装是 AND，但是并不坚持该操作结果，它仅仅是像与操作一样设置标志。

CMP 指令是另一种“操作码幽灵”，该指令同 SUB 的关系和 TEST 同 AND 的关系一样。CMP 指令从第一个操作数中减去第二个操作数，但是并不坚持和存储计算结果到第一个操作数中。该指令仅仅是像减法操作一样设置标志。正如我们已经看到的一样，该指令与条件转移指令相结合可能有些用处。

有一点很重要的事情我们需要牢牢记住：TEST 指令只对查找值为 1 的二进制位有用。如果你需要标识值为零的二进制位，必须通过非指令把每一位转换成相反的状态。非指令把所有值为 1 的二进制位都转换成二进制位 0，把所有的值为 0 的二进制位都转换成 1。当把所有的二进制位 0 都转换成二进制 1 之后，你就可以通过测试某个位是否为 1 来判断它是否为零了。(有时把它画在纸理解起来更直观一些。)

最后，TEST 不能一次可靠地测试两个或多个二进制位的值是否为 1。TEST 不检查位模式是否存在；它只检查一个单独的二进制位 1 是否存在。换句话说，如果需要同时确认第 4 位和第 5 位是否 1，TEST 就不能应对了。

9.4.9 使用 BT 指令查找位 0

如前所述，TEST 有它的局限性：不能判断某个二进制位是否为零。从很早期的 X86 CPU 开始就有 TEST 指令了，但是 386 及较新的处理器中都有另外一条指令，它既能测试位 0 也能测试位 1。BT(位测试)执行的任务非常简单：它将指定的位从第一个操作数中拷贝到进位标志(CF)中。换句话说，如果被选择的是一个二进制位 1，那么 CF 被设置为 1。如果被选择的是一个二进制位 0，那么 CF 就被清零。然后你就可以使用那些需要检验 CF 状态的条件转移指令了。

BT 指令使用起来非常容易。它需要两个操作数：目标操作数是包含被检验位的数；源操作数是你想检测的位的原始编号，从零开始：

```
bt <value containing bit>,<bit number>
```

一旦执行了 BT 指令之后，我们应该立即检验一下 CF 中的值，根据它来进行分支运算。这里有个例子：

```
bt eax,4   ; 检验 AX 中的第 4 位
jnc quit   ; 如果第 4 位=0，我们的任务完成
```

注意，有些内容千万要仔细，尤其是当你已经习惯于使用 TEST 指令时：你不是在创造一个位掩码。对于 BT 指令的源操作数，你正在指出的是某一位的原始编号。上面所显示的字面常数 4 就是所处理位的编号(编号从 0 开始)，而非该二进制位的数值，这是一个至关重要的区别。

有一点也需要注意，那就是如果 CF 没有被设置，我们将进行分支运算，也就是 JNC(如果进位标志未被设置，则跳转)指令所做的事情。

我讨厌在初学者的书籍中过多地讨论编码效率问题，但是这里有一点需要小心：BT 指令的运算过程相当慢——在紧密循环中，我们经常要进行位处理，而在紧密循环中，指令的速度非常重要。使用 BT 指令是可以的，但是如果你位于一个将要执行成千上万甚至上百万次的循环内部，最好慎重考虑一下，是否有其他的更好方法能实现位检测。而快老掉牙的 TEST 指令则要快很多，但是 TEST 只能检测二进制 1。根据应用程序的特点，你也许能够以另外一种更快的方式实

现位 0 检验，或许是通过 SHL 或 SHR 指令将一个值移入 CF，或许是通过非指令将该值进行转换。没有必须遵守的准则，一切都取决于你所做的事情。(这就是为什么在本书中，我并不系统讲解程序优化的原因！)

9.5 保护模式下内存寻址详解

从各个方面来看，我们的生活都变得越来越美好了。我不仅仅是指现代的牙科技术、即插即用的网络，以及 4 核的 CPU。我曾经用汇编语言为老式的 IBM PC 的实模式 8088CPU 进行编码——我至今仍然记得实模式内存寻址。

像 20 世纪 50 年代的牙科技术一样，基于 8088 的实模式内存寻址概括起来就是两个字：痛苦。它像一个具有可怕怪癖的人，条条框框很多，局限性很强，而且浑身上下贴满了“创可贴”，所有一切都发出了这样的垂死吼声： CPU 急需要更多的晶体管！例如，在大多数指令中，内存寻址被限制在 EBX 和 EBP 中，这意味着当同时要在内存中进行几个单独项的寻址时，会需要很多高难度的步骤。现在想起来分段管理，我都不寒而栗。

还好，在最近 20 年中， X86 CPU 已经得到了所需的足够多的晶体管，那些数量众多的、令人愤怒的实模式内存寻址限制已经不复存在。你可以使用任意的通用寄存器来进行内存寻址。你甚至可以通过栈指针寄存器 ESP 来直接对内存进行寻址，而这些事情它的 16 位前身 SP 却只能心有余而力不足了。(在没有经过深思熟虑之前，最好不要轻易改变 ESP 的值，但是 ESP 现在可以参与那些在 16 位实模式下不允许堆栈指针寄存器参与的寻址模式了。)

386 CPU 下的保护模式引入了一个通用内存寻址架构，在该架构中，所有的通用寄存器都可以平等地参与寻址操作。

我已经将其大致地显示在图 9-9 中，它也许是整本书中唯一一个最重要的图解了。建议把它打印出来，挂在计算机旁边的墙上。经常查阅它。内存寻址是汇编语言中最关键的工作。如果对它不理解，其他工作根本无从谈起。

当我第一次研究和理解这幅示意图时，在 16 位 8088 实模式下进行分段内存寻址的痛苦经历依然历历在目。它简直太好了，以至于我很难相信这是真的！但是，它的确是真的！以下是相应的规则：

- 基址寄存器和索引寄存器可以是任意的 32 位通用寄存器，包括 ESP。
- 偏移地址可以是任意的 32 位常数。很明显，0 虽然合法，但是却没有用处。

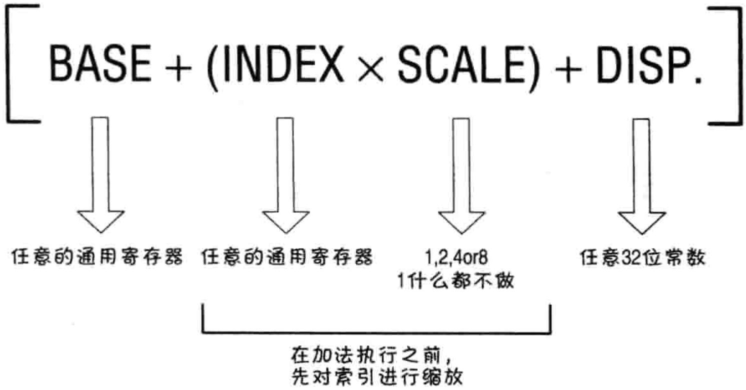


图 9-9 保护模式下内存寻址

- 缩放比例(scale)必须是 1、2、4、8 之中的一个。1 虽然合法，但是不做任何有用的事情，所以从未被使用。
- 在执行加法之前，索引寄存器与缩放比例(scale)相乘。换句话说，不是(base + index) × scale。只有索引寄存器才与 SCALE 相乘。
- 所有的元素都是可选的，而且几乎可以采用任意组合。
- 16 位和 8 位寄存器可能不被用于内存寻址。

最后一点值得我们进一步探讨。通过将图中各元素进行不同的组合，可以实现若干种不同的内存寻址方式。相关示例如表 9-8 所示。

表 9-8 保护模式下的内存寻址方式

方式	举例	描述
[BASE]	[edx]	只有基址
[DISPLACEMENT]	[0F3h] 或 [<变量>]	位移量，或者是字面常数，或者是有符号地址
[BASE + DISPLACEMENT]	[ecx + 033h]	基址+位移量
[BASE + INDEX]	[eax + ecx]	基址+索引
[INDEX × SCALE]	[ebx * 4]	Index * 缩放比例
[INDEX×SCALE + DISPLACEMENT]	[eax * 8 + 65]	索引×缩放比例+位移量
[BASE + INDEX × SCALE]	[esp + edi * 2]	基址+索引乘以缩放比例
[BASE + INDEX × SCALE + DISPLACEMENT]	[esi + ebp * 4 + 9]	基址+索引乘以缩放比例+位移量

9.5.1 有效地址计算

表 9-8 中的每一行都总结了一种 32 位保护模式下的内存寻址方式。除了前两行之外，其他均都涉及一些对括号里的两个或多个表示地址的术语的数学计算。这种数学计算叫作有效地址计算，计算的结果叫作有效地址。其中术语“有效地址”指的是最终被用来读取或者写入的内存地址，不管它以何种方式表示。当指令被执行时，有效地址计算就由该指令来完成。

在基址寻址方式里，有效地址是简单的 32 位值，存放在括号里的通用寄存器中。不涉及任何计算，在源代码中看不到任何文字地址或者符号地址。所以，尽管该指令采用括号之间的寄存器名进行编码，但是当代码执行时，即将发给内存系统是存放在该寄存器内部的地址。

唯一一种有效地址即为指令助记符显示内容的情况就是括号里的字面地址(Literal Address)。这种情况几乎不会发生，因为在汇编时精确地知道一个 32 位数字地址这种情况极不可能发生。

大多数情况下会涉及一些算术运算，例如，在基址+索引这种寻址方式中，当指令被执行时，括号里的两个通用寄存器中的值被相加以产生有效地址。

9.5.2 位移量

在合法地址的若干组成部分中，位移量这个术语实际上最难理解。正如我在上一段中指出的一样，位移量可以是一个字面地址，但是，我在所有的保护模式下汇编编程生涯中从来没有这样做过，也从来没有见过其他人这样做过。当位移量独立出来时，它实际上通常以符号地址的形式出现。在上述说法中，我的意思是指一个已经在.data 段或者.bss 段进行定义的已命名数据项，比如清单 9-1 中的 hexdump1 程序里的 HexStr 变量：

```
mov eax,[HexStr]
```

这里，放入 EAX 中的是当程序被加载到内存里时赋给变量 HexStr 的地址。像所有地址一样，它就是一个数字，但是是在运行时而不是汇编时确定的，就像字面常量地址一样。

当看到在一个地址的括号中好像放有两个位移量时，很多初学者会感到迷惑不解。困惑源于这一个事实：当 NASM 在一个内存引用中看到两个(或多个)常数

值时，它会在编译时将其组合成一个位移量。这里是实现过程：

```
mov eax,[HexStr+3]
```

名为 HexStr 的变量以符号形式引用的地址被简单地与字面常量 3 相加，形成一个单个位移量。位移量的关键特性是它并不位于寄存器中。

9.5.3 基址+位移量寻址方式

一种简单且通用的寻址方式是基址+位移量寻址，我已经在列表 9-1 的 hexdump1 程序中进行了示范。将一个 ASCII 字符插入输出行的指令如下所示：

```
mov byte [HexStr+edx+2]
```

这是一个非常好的例子，里面包含两个位移量，NASM 会将其合并成一个。名为 HexStr 的变量决定了一个数字(HexStr 的 32 位地址)，它很容易与字面常量 2 相加，所以，实际上只有一个基址(EDX)和一个位移量。

9.5.4 基址+索引寻址方式

也许我们最常使用的一种简单寻址方式是基址+索引寻址，通过将括号里的两个通用寄存器的值相加来计算有效地址。我在第 8 章清单 8-2 中的 uppercaser2 程序里示范了这种寻址方式。通过与 20h 相减可以实现把从输入缓冲区中读入的小写字符转换为大写字符：

```
sub byte [ebp+ecx],20h
```

缓冲区的地址已经提前放入 EBP 中，ECX 中的数据是任意给定循环中相对于被处理的字符缓冲区的起始位置的位移。把缓冲区地址与位移量相加，将产生 SUB 指令处理的字符的有效地址。

但是，等一下，为什么不用基址+位移量寻址方式呢？下面的这条指令是合法的呀：

```
sub byte [Buff+ecx],20h
```

然而，如果你还记得那个程序(程序及相关文本很值得重看一遍)，在开始循环之前，我们必须把缓冲区的地址减 1。但是且慢，我们能不能让 NASM 稍作调整，通过与另外一个位移量术语-1 相加来实现呢？实际上我们可以这样做，它同

样会起作用。uppercaser2 程序的核心循环部分如下所示：

```
; 为处理缓冲区这一步骤设置寄存器的值：
    mov ecx,esi                  ; 把需要读入的字节数放入 ecx
    mov ebp,Buff                 ; 将缓冲区地址放入 ebp
; dec ebp                          **我们不再需要这条指令了！**

; 通读缓冲区，将小写字符转换成大写字符：
Scan:
    cmp byte [Buff-1+ecx],61h    ; 检验输入字符是否为小写字母 'a'
    jb Next                      ; 如果 ASCII 码比 'a' 的小，不是小写字母
    cmp byte [Buff-1+ecx],7Ah    ; 检验输入字符是否为小写字母 'z'
    ja Next                      ;如果 ASCII 码比 'z' 的大，不是小写字母
; 此时，我们有了一个小写字母字符
    sub byte [Buff-1+ecx],20h    ; 将给定的大写字母减去 20h
    Next: dec ecx                ; 计数器减 1
    jnz Scan                     ; 如果仍有字符，继续循环
```

最初的 DEC EBP 指令不再是必不可少的了。当程序被加载时，NASM 完成数学运算，有效地址表达式内的缓冲区地址被递减 1。这实际上是一种对这一特定循环的正确编码方式，关于到底应该在第 8 章讲解这部分内容，还是等到我详细讲解内存寻址方式之后再讲解它，我仔细斟酌了很长时间。

有些人觉得“基址+位移量” 容易让人混淆，因为在大多数情况下，位移量包含一个地址，而基址是一个包含了某一数据项相对于那个地址的偏移地址的寄存器。在多数人的经验里，“位移”和“偏移”这两个词非常类似，因此容易混淆。这也是为什么我不在本书中强调各种寻址方式的名字的原因之一，当然，我更不推荐死记硬背这些名字。我们要理解有效地址计算的工作原理，忽略寻址方式的名字。

9.5.5　索引×缩放比例+位移量寻址方式

如果要逐字节地扫描内存中的缓冲区，一般情况下需要使用基址+索引寻址方式，但是如果需要访问一个位于缓冲区中或者表中的数据项，而该数据项不是一个单字节，而是一个字，或者双字，该怎么办呢？这就需要更为强大内存寻址机制。

补充说明一下，数组这个词是我调用缓冲区或表时经常使用的术语。有些其他作者把表叫作数组，尤其是当讨论的上下文是高级语言时，但是所有以下 3 个

条款都对同一定义适用：内存中的一个连续数据项，数据项的大小相同，数据项的内部定义相同。到目前为止，在本书展示的程序中，我们看到的只是非常简单的表和缓冲区，所有行中仅仅包含大小为一个字节的数值。清单表 9-1 中，程序 hexdump1 的 Digits 表就是这样的一张表：

```
Digits: db "0123456789ABCDEF"
```

这是内存中某一行里的 16 个单字节字符。可以采用基址+位移量寻址方式通过以下方式访问 Digits 表中的“C”字符：

```
mov ecx,12
mov edx,[Digits+ecx]
```

如果表里包含的是 32 位的值呢？这样的表很容易定义：

```
Sums: dd "15,12,6,0,21,14,4,0,0,19"
```

修饰符 DD 告诉 NASM，在表 sum 中，每一个数据项都是 32 位双字。字面常量向表中每一个元素插入了一个数字。sum 中第一个元素(这里是 15)的地址就是整个表的地址，它包含在变量 Sums 中。

那么，第二个元素 12 的地址是多少？如何使用汇编编码将其取出？时刻要记住，内存寻址是逐字节而不是逐双字进行的。表中第二个数据项位于相对于该表起始位置的 4 字节偏移处。如果企图通过使用一个 Sums+1 的地址来引用表中第二个数据项，那么你将得到表中第一个双字元素内部的一个字节，而这个字节没有任何用处。

“缩放比例”这个概念闪亮登场了。地址可能会包含缩放比例这一术语，它是一个乘数，可以是 2、4、8 中的任意一个字面常量。(字面常量 1 从技术上来讲是合法的，但是因为缩放比例是一个乘数，所以 1 并不是一个有用的缩放比例值。)索引和缩放比例相乘的结果再与位移量相加，产生有效地址。这就是索引×缩放比例 + 位移量寻址方式。

一般情况下，缩放比例这个术语是表中某一单个元素的大小。如果表包含 2 字节数值，那么缩放比例就是 2。如果表包含 4 字节双字数值，那么缩放比例就是 4。如果表包含 8 字节 4 字数值，那么缩放比例就是 8。

解释这种寻址方式的最好方式是采用图表。我们来观察一下图 9-10 中的地址[DDTable + ECX*4]。DDTable 是一个双字(32 位)表，DDTable 的地址是位移量。ECX 寄存器是索引值，在本例中，它的值为 2，代表你想访问的表元素的编

号。因为它是一个 4 字节双字表，所以缩放比例为 4。

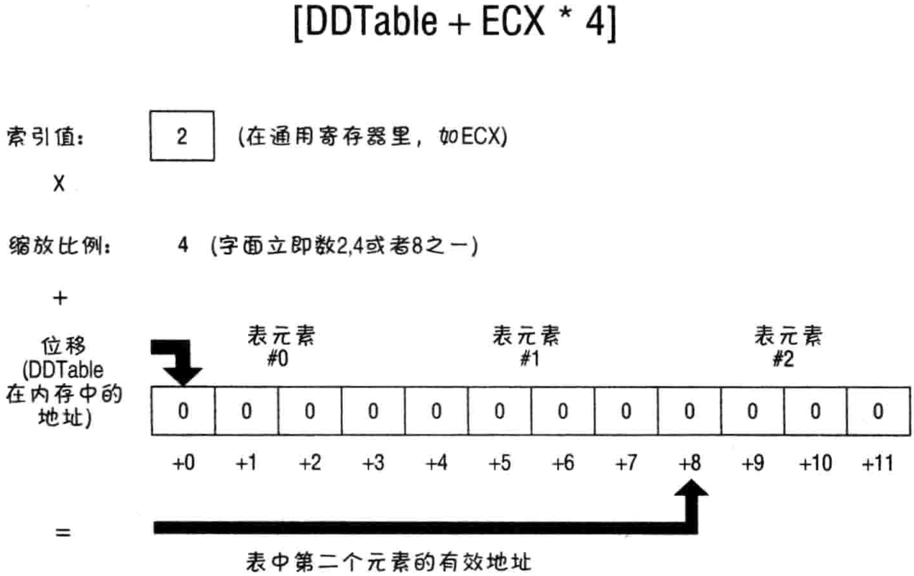


图 9-10　地址缩放工作原理

因为每个表元素的长度都为 4，所以第二个元素相对于表的起始位置的偏移地址为 8。该元素的有效地址通过首先将索引与缩放比例相乘，然后将结果与 DDTable 的地址相加计算而来。看吧，就是这样子！

9.5.6　其他寻址方式

任何包含缩放比例的寻址方式都是按照这种方式工作的。不同之处在于采用什么其他术语来计算有效地址。基址 + 索引 × 缩放比例这种寻址方式将一个缩放后的索引与寄存器中的基址值相加，而不是与一个位移量相加：

```
mov ecx,2              ;索引存放在 ecx 里
mov ebp,DDTable        ; 表地址存放在 ebp 里
mov edx,[ebp+ecx*4]    ; 将选定的元素放入 edx 里
```

你一般不会经常处理像 DDTable 这样的预定义变量地址。有时，表的地址来自其他地方，更多的情况是一张内存里的包含很多子表的二维表，每张子表里包含一些元素。这样的表可通过两步来访问：第一步，推导出内部表在外部表中的

地址；第二步，推导出所需元素在内部表中的地址。

这种类型的二维表的最常见的例子类似于我曾在本书早期版本中介绍过的那个运行在 DOS 下的示例。DOS 下的 25 行×80 字符的文本显存缓冲区就是一张二维表。25 行中的每一行都是一张由 80 个字符组成的表，每一个字符都是由 2 个字节构成的字。(一个字节是 ASCII 码值，另一个字节则指出诸如颜色、下划线之类的属性。)因此，这个缓冲区从整体上来看就是一个由 25 张更小的表构成的总表，每一张小表包含 80 个两字节数值。

那种类型的视频访问系统早就随着 DOS 一起消亡了；Linux 不允许直接访问 PC 显存。然而在 DOS 时代，它可以在很大程度上完成，并且是一个关于二维表的很好示例。

通过按比例缩放，我们可以非常容易地处理包含 2 字节、4 字节、8 字节元素的表。但是如果你的表里包含的是 3 字节元素呢？或者 5 字节元素呢？或者 17 字节元素呢？哎呀，在这种情况下，为了瞄准某一个特定元素，你不得不做一些额外的计算。有效地址计算本身不能完成所有的任务。我已经在清单 9-1 中为你列举了一个关于这种表的例子。那行需要显示的字符串就是一张由 3 字节元素构成的表。每一个元素包含一个空格字符，之后是两个 16 进制位元字符。因为这些元素每个长度为 3 个字符，所以缩放操作不能在指令内部执行，必须单独处理。

这并不困难。缩放 hexdump1 程序中的 HexStr 表中的 3 字节元素可以通过如下指令完成：

```
mov edx,ecx        ; 将字符计数器拷贝到 edx
shl edx,1          ; 通过左移位来实现将计数器*2
add edx,ecx        ; 完成与 3 相乘
```

可以通过一个指令组合来实现将 EDX 中的值与 3 相乘的计算：一条 SHL 指令实现与 2 相乘，之后是一条加法指令，实现把索引值的一份拷贝与移位后的索引值相加，有效地实现了将原始值与 3 相乘。

以其他比例缩放索引值可采用以同样的方式完成。缩放 5 倍可以通过将索引值左移 2 位，进而实现与 4 相乘，之后再与索引值的另一份拷贝相加，进而完成与 5 相乘。概括地讲，为了将一个索引值缩放 X 倍：

(1) 找出比 X 小的那个最大的 2 的幂数。

(2) 将索引值左移那个 2 的幂数位。

(3) 不止一次地将原始索引值的一份拷贝与移位后的那个拷贝相加，直到完

成与 X 相乘。

例如，如果 X 为 11，则缩放计算可以通过如下所示代码完成：

```
mov edx,ecx     ; 将索引拷贝到 edx 中
shl edx,3       ; 通过左移 3 位实现与 8 相乘
add edx,ecx     ; 执行三次与索引拷贝值相加的第一次加法
add edx,ecx     ; 执行三次与索引拷贝值相加的第二次加法
add edx,ecx     ; 执行三次与索引拷贝值相加的第三次加法
```

这种方法对于缩放比例相对较小的情况非常有效；一旦超过 20，将会用到很多条加法指令。从这一点来讲，解决方案不是为了计算缩放比例，而是为了查找专门为某一给定缩放比例而定义的比例值。例如，假设表中每个元素长度都为 25 个字节。你可以定义一张 25 的倍数的表：

```
ScaleValues: dd 0,25,50,75,100,125,150,175,200,225,250,275,300
```

对于每一项大小为 25 个字节的表而言，为了得到索引值为 6 的表项的地址，我们可以通过如下方式在表中查找 6×25 的结果：

```
mov ecx,6
mov eax,[ScaleValues+ecx*4]
```

现在，EAX 中包含了第 6 个元素的第一个字节的有效地址，元素计数(通常)从 0 开始。

9.5.7 LEA：最机密的数学机器

稍等片刻(向他们在午夜电视中常说的那样)，后面的内容更精彩。在 X86 体系结构中，有一条最古老但是从某个方面来讲又最神奇的指令：LEA，加载有效内存。从表面上看，它所做的事情非常简单：根据括号里给出的源操作数来计算有效地址，然后把该地址加载到它的目标操作数所指定的任意的 32 位通用寄存器中。回过来看看上一自然段，以及那条在 ScaleValues 表中查找索引值为 6 的元素的 MOV 指令，它必须首先计算出索引值为 6 的那个数据项的有效地址。然后使用这个地址访问内存。

如果你想要把这个存放在寄存器中的地址存起来，以便将来使用，该怎么办呢？这就是 LEA 指令做的事情。这里是一条正在工作中的 LEA 指令：

```
lea ebx,[ScaleValues+ecx*4]
```

这里发生的事情是：CPU 根据方括号中给定的内容计算有效地址，然后将那个地址加载到 EBX 寄存器中。记住，表中的每一个单独成员并没有标号，因此不能被直接引用。LEA 能够让你计算表中任意元素的有效地址(或任意可计算的地址！)，然后将该地址放入寄存器中。

从本质上讲，这是非常有用的，但是，LEA 还有一个特殊的作用：进行无须执行移位、相加或者迟钝的 MUL 操作的快速数学运算。如果你还记得的话，应该知道在 hexdump1 程序中有一个通过一条移位指令和一条加指令实现与 3 相乘的运算：

```
mov edx,ecx     ; 把字符计数器拷贝到 edx
shl edx,1       ; 通过左移位实现将指针与 2 相乘
add edx,ecx     ; 完成与 3 相乘
```

前面这些指令是可以实现相应功能的，但是看一下我们能使用什么其他指令完成同样的事情：

```
mov edx,ecx             ; 把字符计数器复制到 edx 中
lea edx,[edx*2+edx]     ; 将 edx 乘以 3
```

事实上，这样做不仅通常比移位指令和加法指令的组合更迅速，而且也可以从源代码中非常清晰地看出到底正在进行哪一种运算。EDX 中的值也许实际上并不是某些东西的合法地址，但是这一事实并不重要。*LEA 并不设法引用它计算出来的地址*。它只是根据括号里的内容完成数学运算，然后再将结果存入目标操作数。任务完成。不接触内存，也不影响标志位。

当然，你被限制在那些能够对有效地址进行的数学运算里，但是稍微超出这个范围一点点，那就是当放入一个常数的时候，你也可以实现将任意的通用寄存器与 2、3、4、5、8 和 9 相乘！它不是任意的数学运算，但是在汇编中，经常会有与 2、3、4、5、8 和 9 相乘的情况，你可以把 LEA 和移位指令以及加法指令组合在一起，完成更为复杂的数学运算。你还可以在一行中多次使用 LEA 指令。两个连续的 LEA 指令可以实现把一个值与 10 相乘，实际上这一点很有用：

```
lea ebx,[ebx*2]         ; 把 ebx 与 2 相乘
lea ebx,[ebx*4+ebx]     ; 把 ebx 与 5 相乘，以求取与 10 相乘的最终结果
```

有些人将其看成小儿科的把戏，但是在使用 X86 汇编语言工作的所有日子里，我还从来没有发现过它的不足之处。在打算通过五六条指令来完成某个乘法运算之前，看看能否通过两三条 LEA 指令来代替它。LEA 在一个机器周期里完

成它的工作，X86 数学运算中没有比它再快的了！

9.5.8 16 位寄存器的负担

在保护模式下，对 32 位寄存器进行分级管理，这存在着一定的负面效果：使用 16 位通用寄存器 AX、BX、CX、DX、SP、BP、SI 和 DI 将会使运行速度减慢。由于 32 位寄存器规则，使用 16 位寄存器被认为是一种特例，它会增加汇编语言产生的操作码的尺寸，减慢程序代码的向下执行速度。现在，通过“使用”这个词，我的意思在源代码中进行显式引用。例如，AX 寄存器仍然位于 CPU 的硅芯片的内部(作为更大的 EAX 寄存器的一部分)，简单地把数据放入其中并不会使速度减慢。但是，只要将 AX 作为某个操作码的操作数，以这种方式将数据放入 AX 中，执行速度肯定会减慢。以下的语法将产生一个执行速度比较慢的操作码：

```
mov ax,542
```

我们也可以采用如下的操作码完成同样的事情，而且 NASM 产生的操作码的执行速度会更快：

```
mov eax,542
```

直到现在才开始提到这些内容，是因为我认为它是一个高级话题：你不能不会走就想跑，在还没有充分地理解什么会使代码运行速度快些，什么会使代码运行速度慢些之前，企图优化代码已被证实只会给你带来困惑和失望。在程序中三三两两地引用 16 位寄存器并不会使这个程序显著变慢。我们需要避免的是，不要在一个将会被执行几千次甚至成千上万次(甚至更多！)的紧密循环中使用 16 位寄存器引用。

在某些情况下，使用 8 位和 16 位寄存器都是十分有必要的——例如将一个 8 位或 16 位数值写入内存时，NASM 不允许你这样做：

```
mov byte [ebx],eax
```

BYTE 标识符使第一个操作数成为一个 8 位操作数，NASM 将会发出“操作数尺寸不匹配”的牢骚。如果你需要将一个单独的 8 位数值(如一个 ASCII 字符)写到内存中，你需要将这个字符放入一个 8 位寄存器，就像这样：

```
mov byte [ebx],al
```

这会产生一个中等慢的操作码，但是没能绕过其他指令。记住，尤其是对于现代的 CPU 而言，单独的操作码的代码性能已经淹没在 CPU 的一些机制(如高速缓存，超线程，预取指技术等等)的性能之中。一般情况下，统计表明，使用 32 位寄存器更可能使代码运行快些，但是三三两两地使用 16 位或 8 位寄存器引用并不会带来很大的区别，除非在某些特定情况下(如在紧密循环内部)，即便如此，性能“攻击”依然很难预测，而且几乎不可能被量化。

只要简单记住：凡是能使用 32 位寄存器的地方就使用 32 位寄存器，不要再为这个问题而苦恼了。

9.6 字符表转换

有一种类型的查表操作，它(或者曾经是)非常普遍以至于 Intel 的工程师专门为它烤制了一条单独的指令到 X86 体系结构中。这种类型的表查找操作就是我在第 8 章结尾的潜台词：字符转换。在 20 世纪 80 年代早期，我需要以各种方式来转换字符序列，其中最简单的一种是把所有的字母从小写转换成大写。在第 8 章里，我创建了一个简单的程序，它一次审查文件的一个缓冲区，读入字符，将所有的小写字符转换成大写，然后再将它们写回到一个新文件里。

转换本身很简单：通过 ASCII 表来获取所有的大写字母和它们相应的小写字母之间的关系，我们可以通过简单地把这个字母的 ASCII 码减去 20h(32)来实现将其转换成大写字母。那样做是可靠的，但是在某种意义上来讲是一种特例。仅仅是恰巧在 ASCII 码表里，小写字母的 ASCII 码比大写字符的 ASCII 码大 32。如果需要把所有的“竖线” (ASCII 124)字符转换成惊叹号，该怎么做呢？(我必须这样做，因为有一种类型的傻瓜主机根本不“接纳”竖线。)可以为每一种必须处理的情况单独编写特定的代码，或者，也可以使用转换表。

9.6.1 转换表

转换表是一种特殊类型的表，它按照如下方式进行工作：建立一张数值表，并为每一个必须被转换的可能值创建一个表项。采用数字(或被当成数字的字符)作为该值在这张表中的索引。索引指定的表中位置是一个用于替换被用作索引的原值的数字。简言之，原值被当成该值在查找表中的索引，然后根据索引在表中找到一个新的值来替代原来的值，这样就实现了将原来的值转换成一个新值。

我们以前曾经在清单 9-1 的 hexdump1.asm 程序里这样做过。回想一下 Digits 表：

```
Digits: db "0123456789ABCDEF"
```

这是一张转换表，尽管那时我没有这样称谓它。其中的思想，如果你还记得的话，是将一个 8 位二进制数分成两个 4 位的半字节，然后把那些 4 位值转换成代表十六进制位元的 ASCII 字符。那时的焦点是通过按位逻辑运算将字节分成两个半字节，但是，那时也用到了转换。

转换通过以下三条指令完成：

```
mov al,byte [esi+ecx]         ; 从输入缓冲区中取出一个字节，放入 al
and al,0Fh                    ; 屏蔽掉除了低半字节之外的所有位
mov al,byte [Digits+eax]      ; 查找与低半字节相等的字符
```

第一条指令从输入缓冲区中加载一个字节到 8 位的 AL 寄存器中。第二条指令屏蔽 AL 中除了低半字节之外的所有位。第三条指令读取内存：它将 AL 中的值当成 Digits 表的索引，然后取回表中的第 AL 个表项的值。(必须通过括号中的 EAX 来实现，因为 AL 不能参与有效地址计算。只需要记住，AL 是 EAX 寄存器中的最低位字节。)如果 AL 中的值为 0，那么有效地址计算把 0 加到 Digits 的地址上，带回第 0 个表条目，即 0 的 ASCII 字符。如果 AL 中的值为 5，那么有效地址计算把 5 加到 Digits 的地址上，带回第 5 个表条目，即 5 的 ASCII 字符。以此类推，它能找到所有的 16 个可能用于表示 4 位的半字节的值。从根本上来讲，这些代码用于将一个数字转换为相应的 ASCII 字符。

只有 16 个可能的 16 进制数字，所以 hexdump1 中的转换表值需要 16 个字节长。一个字节包含足够的位来表示 256 个不同的值，所以我们如果想要转换字节大小的值，需要一张具有 256 个条目的表。从技术上讲，ASCII 字符序列只使用前 128 个值，但是正如本书前面的章节所说的那样，“高”128 个值通常被分配给特殊的字符，如非英语字母、#字符、数学符号等。字符转换的一个最常见的用途是把值大于 128 的字符转换成某些值小于 128 的字符，以避免在那些不能处理扩展 ASCII 值的老式系统中发生混乱。

如下所示的表在汇编语言程序中非常容易定义：

```
UpCase:
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
```

```
db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
db 60h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,7Bh,7Ch,7Dh,7Eh,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
```

表 UpCase 被定义成 16 行，每一行由 16 个单独的十六进制值组成。表被分成 16 行仅仅是为了屏幕上或者打印页上的可读性，并不会影响 NASM 在输出中产生的二进制表。一旦以二进制形式表示，它是位于一行中的 256 个 8 位值。

这里是一个快速的句法注释：当定义表(或者任何包含多个预定义值的数据结构)时，使用逗号分隔一个定义中的数值。在表中每行 DB 定义的结尾没有必要使用逗号。每一个 DB 定义都是分开且独立的，但是因为它们在内存中是毗邻的，所以我们可以把这 16 个 DB 定义看成一张简单的表。

任何转换表都可被看作表达了一种或者多种管理转换过程中发生的各种事情的“规则”。上面的 UpCase 表表达了以下转换规则：

- 所有的小写 ASCII 字符被转换成大写。
- 所有小于 127 的非小写字母的可打印 ASCII 字符被转换成它们自己。(严格来讲，它们并没有被丢下不管，而是转换成了同样的字符。)
- 所有 ASCII 码值位于 127～255 之间的“高位”字符都被转换成 ASCII 空格字符(32 或者 20h)。
- 除了 9 和 10 之外，所有不可打印 ASCII 字符(总的来说，值在 0～31 之间，还有 127)被翻译成空格。
- 字符值 9 和 10 (tab 和 EOL) 被转换成它们本身。

对每一条数据项来说都不错，不是吗？(只需想象一下，如果纯粹使用机器指令完成所有那些令人烦恼的事情的话，将会有多少工作需要做！)

9.6.2 用 MOV 或者 XLAT 进行转换

那么如何使用这张 UpCase 表呢？很简单：

(1) 将即将被转换的字符加载到 AL 中。

(2) 通过使用 AL 作为基址，UpCase 作为位移量，创建一个内存引用，然后通过 MOV 指令将该内存引用处的字节存储到 AL 中，替换掉那个作为基址使用的原值。

MOV 指令如下：

```
mov al,byte [UpCase+al]
```

但是有一个问题：NASM 不会允许你这么做。AL 寄存器不能参与有效地址运算操作，所有其他的 8 位寄存器也不能参与。让我们进一步研究一下 XLAT 指令。

XLAT 指令是被硬编码的，用于以特定的方式使用特定的寄存器。它的两个操作数是隐式的：

- 转换表的地址必须在 EBX 中。
- 被转换的字符必须在 AL 中。
- 转换之后的字符会被返回 AL，替代 AL 中的原始字符。

通过以上方式设置好寄存器之后，XLAT 指令就可以独自使用了：

```
Xlat
```

这里我诚实地交代：XLAT 再也不能像以前那样处于绝对获胜地位了。在 32 位保护模式下，同样的事情也可以采用以下指令完成：

```
mov al,byte [UpCase+eax]
```

这里只有一个“陷阱”：你必须清除 EAX 高 24 位的所有剩余值，否则你很可能意外地访问远远超出转换表的边界的索引。XLAT 指令仅仅把 AL 当作索引来用，忽略其他所有 EAX 中的可能值。在加载即将被转换的值到 AL 中之前，一定要通过一条简单的 XOR EAX,EAX 指令或者 MOV EAX,0 将 EAX 清零。

事实上，在给定了使用 AL 和 EBX 的要求下，XLAT 指令只是一副淡水彩画，但是，我一直试图在这里介绍的，是一个通过表来实现字符转换的更大主题。清单 9-2 实现了这一主题。这里所展示的程序跟清单 8-2 中的 uppercaser2 程

序的功能完全相同：它强制把一个输入文件的所有小写字符转换成大写，然后再将其写入一个输出文件。我没有将其叫作“uppercaser3”，因为它是一个通用字符转换程序。在这个特定程序里，它把所有的小写字符转换成大写，但是这只是 UpCase 表所表示的规则之一。修改这张表即可修改转换规则。你可以把任意一个值转换成 256 个值中的任意一个，甚至可以批量转换，即：将一个字节能够表示的全部值转换成 256 个不同值。

清单 9-2：xlat1.asm

```
; 可执行程序名：XLAT1
; 版本：1.0
; 创建日期 : 2/11/2009
; 最后修改日期 : 4/5/2009
; 作者 : Jeff Duntemann
; 描述：用 NASM 2.05 实现的一个运行在 Linux 下的简单汇编程序
; 示范了如何使用 XLAT 指令来修改文本流
;
; 使用如下命令生成：
; nasm -f elf -g -F stabs xlat1.asm
; ld -o xlat1 xlat1.o
;

SECTION .data          ; 包含已初始化数据的段

        StatMsg: db "Processing...",10
        StatLen: equ $-StatMsg
        DoneMsg: db "...done!",10
        DoneLen: equ $-DoneMsg

; 下面的转换表把所有的小写字母转换成大写。
; 它还把所有的不可打印字符转换成空格。
; 除了 LF 和 HT 之外。
    UpCase:
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
    db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
    db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
    db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
    db 60h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
    db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,7Bh,7Ch,7Dh,7Eh,20h
```

```
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
; 下面的转换表“致力于”将所有的可打印字符转化成它们本身
; 除了 LF 和 HT 之外，将所有的不可打印字符转换成空格。
; 你可以修改它，使之能够把你想转换的任意字符转换成任意你想要的结果
    Custom:
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
    db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
    db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
    db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
    db 60h,61h,62h,63h,64h,65h,66h,67h,68h,69h,6Ah,6Bh,6Ch,6Dh,6Eh,6Fh
    db 70h,71h,72h,73h,74h,75h,76h,77h,78h,79h,7Ah,7Bh,7Ch,7Dh,7Eh,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
    db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h

SECTION .bss                    ; 包含未初始化数据的段

    READLEN equu 1024           ; 缓冲区的长度
    ReadBuffer: resb READLEN    ; 文本缓冲区本身

SECTION .text                   ; 包含代码的段

global _start                   ; 连接程序需要利用它找到入口点！

_start:
    nop                         ; 这个无操作指令让 gdb 感到非常高兴……

; 通过标准错误 stderr 显示信息：“我正在工作……”：
    mov eax,4                   ; 指定 sys_write 调用
```

```
    mov ebx,2                     ; 指定文件描述符 2: 标准错误
    mov ecx,StatMsg               ; 传递该信息的偏移地址
    mov edx,StatLen               ; 传递该消息的长度
    int 80h                       ; 进行内核调用

; 从标准输入中读入满满一缓冲区文本:
read:
    mov eax,3                     ; 指定 sys_read 调用
    mov ebx,0                     ; 指定文件标识符 0: 标准输入
    mov ecx,ReadBuffer            ; 传递需要读取的缓冲区的偏移地址
    mov edx,READLEN               ; 传递一次运行需要读入的字节数
    int 80h
    mov ebp,eax                   ; 复制 sys_read 的返回值，并妥善保管
    cmp eax,0                     ; 如果 eax=0，则 sys_read 到达了文件结尾
    je done                       ; 如果相等则跳转(通过与 0 比较)

; 为下面的转换步骤设置寄存器:
    mov ebx,UpCase                ; 将表的偏移地址存入 ebx
    mov edx,ReadBuffer            ; 将缓冲区的偏移地址存入 edx
    mov ecx,ebp                   ; 将缓冲区中的字节数存入 ecx

; 使用 xlat 指令转换缓冲区中的数据:
; (注意: 添加注释的指令跟 XLAT 做同样的事情)
; 去掉注释，然后把 XLAT 注释掉，试试看!
translate:
    ; xor eax,eax                 ; 清除 eax 的高 24 位
    mov al,byte [edx+ecx]         ; 加载字符到 AL 中已备转换
    ; mov al,byte [UpCase+eax]    ; 通过表来转换 AL 中的字符
    Xlat                          ; 通过表来转换 AL 中的字符
    mov byte [edx+ecx],al         ; 把转换后的字符存回缓冲区
    dec ecx                       ; 把字符计数器减 1
    jnz translate                 ; 如果缓冲区里还有字符，重复执行。

;把装满了转换后的文本的缓冲区写入标准输出:
write:
    mov eax,4                     ; 指定 sys_write 调用
    mov ebx,1                     ; 指定文件描述符 1: 标准输出
    mov ecx,ReadBuffer            ; 传递缓冲区的偏移地址
    mov edx,ebp                   ; 传递字节数到缓冲区中
    int 80h                       ; 进行内核调用
    jmp read                      ; 回去继续循环，加载另一个装满数据的缓冲区

; 通过标准错误显示信息"我做完了":
```

```
done:
    mov eax,4                         ; 指定 sys_write 调用
    mov ebx,2                         ; 指定文件描述符 2：标准错误
    mov ecx,DoneMsg                   ; 传递该信息的偏移地址
    mov edx,DoneLen                   ; 传递该信息的长度
    int 80h                           ; 进行内核调用

; 全部完成！让我们结束这场“聚会”：
    mov eax,1                         ; 退出 Syscall
    mov ebx,0                         ; 将 0 作为返回值
    int 80H                           ; 进行内核调用
```

我已经为这个程序添加了另外一张表，为了你做实验使用。custom 表代表了如下规则：

- 所有小于 127 的可打印 ASCII 字符被转换成它们自己。(严格来讲，它们并没有被丢下不管，而是转换成了同样的字符。)
- 所有 ASCII 码从 127～255 的“高位”字符都被转换成 ASCII 空格字符(32 或者 20h)。
- 所有不可打印 ASCII 字符 (总的来说，值在 0～31 之间，还有 127) 被翻译成空格，除了 9 和 10 之外。
- 字符值 9 和 10 (tab 和 EOL) 被转换成它们本身。

从根本上来讲，它把所有的可打印字符(再加上 tab 和 EOL)丢掉不管，把所有的其他字符转换成 20h，即空格字符。你可以将程序中的标号 UpCase 替换为 Custom，再对 Custom 表做些修改，然后进行试验。比如，把那些恼人的竖线转换成感叹号，把所有的“Z”转换成“Q”。通过改变表中的内容来实现规则的变换。代码丝毫都不用变！

9.7　用表来代替计算

计算机系统的标准化使得字符转换不再像以前那样普遍，但是转换表在某些其他领域可以相当的有用。其中之一就是执行快速数学运算。考虑一下以下的表：

```
Squares: db 0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225
```

这里没有什么神秘之处：Squares 是一张 0-15 之间的数字的平方表。如果在某个计算中需要用到 14 的平方，你可以使用 MUL 指令，但是它的运行速度非常

慢，而且需要两个通用寄存器；或者你也可以简单地从 Squares 表中取出结果：

```
mov ecx,14
mov al,byte [Squares+ecx]
```

瞧！EAX 现在就包含了 14 的平方。你可以用 XLAT 完成同样的功能，虽然它要求使用特定的寄存器。还要记住，XLAT 被限制在 8 位量内。上面显示的 Squares 表是 XLAT 所能使用的最大的平方表，因为下一个平方值(16)是 256，它不能用 8 位表示。

建立一张每个条目都是 16 位的平方值的查找表能够让你包含 0～255 之间的所有整数的平方。如果查找表的每一个条目都是 32 位，你就能够包含 0～65 535 之间的所有整数的平方——但是那将是一张规模非常可观的表！

在本书中，我没有足够的篇幅来深入讲解浮点型数学运算了，但是，使用表来查找一些诸如平方根之类的值曾经使用得非常频繁。近几年来，在 CPU 中直接包含数学运算的处理器使得这类技术不再像以前那样具有竞争力了。但是，当面对一个整数运算的挑战时，你应该时刻在脑中想着使用查找表的可能性。

第 10 章

分治

使用进程和宏来对抗程序的复杂性

复杂性足以杀掉一个程序。这是我作为程序员以来曾经获得的第一个教训，而且 30 多年来，这个教训一直纠缠着我。

所以请听好：有一种编程语言叫作 APL(“A Programming Language”的首字母缩写，多么聪明)，它不止拥有一点点火星人习性。APL 是我曾经学习过的第一种计算机语言(在一台主流的 IBM 大型机上)，而且在学习它的过程中，我学到的东西比 APL 本身包含的知识还要多一点。

APL 使用一种非常紧凑的表示法，且包含自己独有的、几乎与我们熟悉的 ASCII 字符完全不同的字符集合。该字符集合包含许多奇怪的小符号，每个符号具有一些令人震惊的能力，例如：转置矩阵。就一行代码能做的事情而言，APL 能够比任何一种我曾学过的其他语言做更多的事情。这些奇怪符号系列的组合和趋近于令人难以觉察的紧凑表示法使得阅读和记忆 APL 中的一行代码实际上所做的事情变得非常困难。

那是 1977 年。已经掌握了(或者说自认为掌握了)整个符号库后，我打算编写一个文本格式化程序。该程序能够调整左右对齐，标题居中，以及做一些在我们今天看来理所当然但是在 70 年代却不同寻常的其他事情。

该程序在一个星期内增长到超过大约 600 行扭曲的小 APL 符号。我开始使用它工作，而且它工作得非常好——只要我没有尝试格式化一个超过 64 个字符宽的列。然后一切开始出现混乱。

情况不妙。我将整个程序打印了出来，坐下来仔细地调试。然后，我意识到了一种恐怖的下沉感觉：在完成了程序的最后一部分之后，我已经不再知道它的第一部分的工作原理了。

这个特殊的 APL 符号集只是问题的一部分。我很快认识到，我犯下的最重

要的错误，是将全部内容写在了一个整个的 600 行代码块中。没有功能分割，没有任何东西能够表明任何由 10 行组成的代码块儿是在尝试完成什么任务。

火星人胜利了。我做了唯一一件可能的事情：我放弃了那个程序，硬着头皮接受我的文本中页边线参差不齐的现状。复杂性是程序的致命之伤，这是我的血泪教训。它不仅对于汇编语言而言是正确的，而且适用于 Java、C、Pascal 或者曾经出现过的任何其他编程语言。现在，你已经能够编写相当复杂的汇编程序了，你最好学会如何管理这种复杂性，否则你会发现自己不得不放弃大量代码，仅仅是因为记不起来或者无法推算出它的工作原理。

10.1 盒子里面的盒子

在编程过程中，管理复杂性是一项非常巨大的挑战。其关键技巧听起来有点像东方神秘主义，但它实际上不过是一个来自生活的观察结果：任何行动的里面都包含着一个个更小的行动。拿日常活动来说吧。刷牙时，会执行下列操作：

- 拿起牙膏管。
- 拧开牙膏管上面的盖子。
- 将盖子放在洗涤槽的柜台上。
- 拿起牙刷。
- 从管中挤出牙膏到牙刷上。
- 将牙刷放入嘴里
- 用力地来回刷牙。

等等。刷牙时，你执行了这些行为中的每一个。但是，当想了解其中的顺序时，你没有贯穿整个列表。你只是想起了一个单个的概念，叫作“刷牙”。

此外，当思考我们叫作“早上起床”的行动背后的内容时，也可以组建一个如下所示的活动列表：

- 关闭收音机闹钟。
- 爬下床。
- 穿上睡袍。
- 让狗出来。
- 做早餐。
- 吃早餐。
- 刷牙。

- 剃须。
- 淋浴。
- 穿衣服。

刷牙当然是在列表上，但是在你称为“刷牙”的活动内部是如前所述的一整个规模较小的行动的列表。在上述列表中显示的大部分活动也可以这样说。例如，要想准备一顿合理的早餐，需要多少个单独的行为？然而，如果纵观“早上起床”这一行动中的每一个较小的阶段，我们都可以通过由一串小的以及更小的行动所构成的较大行动来描述，而不必辛辛苦苦地跟踪每一个行动。

我所描述的是“中国盒子”式的与复杂性的作战方法。“早上起床”这一行动涉及数百个小行动，所以我们将这个大规模行动分割成连贯的块儿，并将这些块儿设置成小的概念盒。“制作早餐”在一个盒子里，“刷牙”在另一个盒子里，“穿衣服”还有一个盒子，以此类推。仔细观察任何一个单个的盒子，你都会发现，其内容还可以再细分为更多的盒子，而那些更小的盒子还能细分成甚至还要小的盒子。

这个过程不会(也不能)永远持续下去，但是只要需要，它就应该继续，直到满足这一标准：任何一个盒子的内容仅需一点点仔细研究就应该能被理解。每个单个的盒子不应该包含比较微妙或者较大块儿的内容，而且也不应让人们花费几个小时的时间盯着它，绞尽脑汁才能推算出来。

作为代码盒子的过程

我在编写 APL 文本格式化实用程序时所犯下的错误，就是将整个 600 行代码扔到一个被标识为“文本格式化实用程序”的巨大盒子里。

在编写软件期间，我本应该时刻注意那些共同工作以完成某一可识别任务的代码序列。当看到这样的序列时，我本该把它们设置为过程，并为每一个过程赋予一个描述性名称。那样的话，每个序列就会拥有一个记忆标号(即：它的名字)以标识该序列的功能了。如果实现调整一行文字位置的操作使用了十条语句，那么这十条语句应该被命名为 JustifyLine，等等。

施乐公司的传奇 APL 程序员吉姆·邓恩后来告诉我，APL 过程不应该编写得过长，以至于不能在一个 25 行的终端屏幕上显示完全。“如果超过 25 行，你在一个过程中所做的事情就太多了。所以，要把它分开。”他说。从此以后，每当使用 APL 工作时，我都坚持那条睿智的经验规则。“火星人”仍然不时罢工，但是当它们那样做的时候，我们所面对的损失已经不再是全军覆没了。

所有当今常用的计算机语言都执行这种或那种形式的过程，汇编语言也不例外。一个汇编语言程序可能拥有许多个过程。事实上，对于可以在一个程序中包含多少个过程没有任何限制，只要所有过程所包含的代码加在一起的字节总数，加上它们使用的所有数据，不超过 Linux 分配给单个用户空间程序的两千兆不断变化的字节即可。

不用说了，那是很多的代码。即使是最大的商业应用程序，如 OpenOffice，也没有那么大。无论编写的汇编语言程序多么复杂，我们都可以通过过程来进行管理。

让我们首先从一个正在应用中的过程的示例开始。仔细阅读程序清单 10-1，让我们来看看是什么使它工作，以及是(更重要的是)什么有助于它仍然保持可理解性。

清单 10-1：hexdump2.asm

```
; 可执行程序名   : hexdump2
; 版本 : 1.0
; 创建日期       : 4/15/2009
; 最后修改       : 4/20/2009
; 作者           : Jeff Duntemann
; 描述           : 一个简单的十六进制转储实用工具，演示了汇编语言过程的使用。
;
; 使用以下命令生成该程序 :
;    nasm -f elf -g -F stabs hexdump2.asm
;    ld -o hexdump2 hexdump2.o
;

SECTION .bss                 ; 包含未初始化数据的段

   BUFFLEN EQU 10
   Buff resb BUFFLEN

SECTION .data                ; 包含已初始化数据的段

; 这里，我们使用一个由两个部分组成的简单数据结构，
; 实现一个十六进制转储实用用具的文本行。
; 第一个部分显示了 16 个字节的、中间用空格隔开的十六进制数字。
; 紧跟在后面的是一个由 16 个字符组成的行，二者之间通过竖线字符分隔。
; 因为这两个部分是相邻的，
; 所以可以被单独引用或者作为一个连续的单元来引用。
; 记住，如果要将 DumpLin 分开使用，在将其发送到 Linux 控制台之前，
```

```
; 必须追加一个 EOF。

DumpLin: db " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "
DUMPLEN EQU $-DumpLin
ASCLin: db "|................|",10
ASCLEN EQU $-ASCLin
FULLLEN EQU $-DumpLin

; HexDigits 表用来把数字值转换为它们的十六进制等值。
; 通过一个没有缩放的半字节来进行索引： [HexDigits + eax]
HexDigits: db "0123456789ABCDEF"

; 此表用于 ASCII 字符翻译，实现将其翻译成该十六进制转储行的 ASCII 部分
; 通过使用 XLAT 或者普通的内存查找。
; 所有的可打印的字符被翻译为它们本身。
; 高 128 个字符被转换为 ASCII 的句号(2Eh)。
; 低 128 个字符中的不可打印字符也被转换为 ASCII 句号，例如：字符 127。
DotXlat:
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
        db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
        db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
        db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
        db 60h,61h,62h,63h,64h,65h,66h,67h,68h,69h,6Ah,6Bh,6Ch,6Dh,6Eh,6Fh
        db 70h,71h,72h,73h,74h,75h,76h,77h,78h,79h,7Ah,7Bh,7Ch,7Dh,7Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh

SECTION .text       ; 包含代码的段

;-----------------------------------------------------------------
; ClearLine:   将一个十六进制转储行字符串清零，即将其变为 16 个 0 值
; 更新日期:     4/15/2009
; 输入参数:     没有。
; 返回值:       没有。
```

```
; 修改:          没有。
; 调用:          DumpChar
; 描述:          通过 16 次调用 DumpChar 过程，每次传递给它参数 0
; 这个十六进制转储行字符串被清除为二进制 0。

ClearLine:
        Pushad                     ; 保存主调程序的所有通用寄存器
        mov edx,15                 ; 我们将进行 16 次，从 0 开始计数
.poke:  mov eax,0                  ; 告诉 DumpChar 插入一个'0'
        call DumpChar              ; 将'0'插入到十六进制转储字符串中
        sub edx,1                  ; DEC 并不影响 CF!
        jae .poke                  ; 如果 EDX >= 0 ,回去继续循环。
        popad                      ; 恢复所有主调程序的通用寄存器。
        Ret                        ; 返回

;------------------------------------------------------------------
; DumpChar:      插入一个值到十六进制转储字符串中。
; 更新日期: 4/15/2009
; 输入参数: 传递一个即将被插入的值到 EAX 寄存器中。
; 传递这个值在该行中的位置(0-15)到 EDX 寄存器中。
; 返回值:        没有。
; 修改:          EAX, ASCLin, DumpLin
; 调用:          没有。
; 描述:          传递到 EAX 寄存器中的值将其放在十六进制转储部分，
; 又放在 ASCII 部分，
; 其位置为传递给 EDX 的值，
; 当它不是一个可打印字符时，用一个空格来表示。

DumpChar:
        push ebx          ; 保存主调程序的 EBX 寄存器
        push edi          ; 保存主调程序的 EDI 寄存器
; 首先，我们将输入的字符插入到转储行的 ASCII 部分
        mov bl,byte [DotXlat+eax]    ; 将不可打印字符翻译成 '.'
        mov byte [ASCLin+edx+1],bl   ; 写入 ASCII 码部分
; 接下来，我们把与插入字符等值的十六进制值
; 插入到转储行的十六进制部分:
        mov ebx,eax                  ; 保存输入字符的第二个副本
        lea edi,[edx*2+edx]          ; 计算字符串行中的偏移量(EDX*3)
; 查找低半字节的字符，并将其插入到字符串:
        and eax,0000000Fh            ; 屏蔽掉除低半字节以外的所有位
        mov al,byte [HexDigits+eax]  ; 查找该半字节的等值字符。
        mov byte [DumpLin+edi+2],al  ; 将这个等值字符写入字符串行。
```

```
; 查找高半字节的字符，并将其插入到字符串：
        and ebx,000000F0h               ; 屏蔽掉除第二低的半字节之外的所有位
        shr ebx,4                       ; 将字节的高四位移到低四位
        mov bl,byte [HexDigits+ebx]     ; 查找与该半字节的等值字符。
        mov byte [DumpLin+edi+1],bl     ; 将这个等值字符写入字符串行。
;任务完成！让我们"回家"：
        pop edi                         ; 恢复主调程序的 EDI 寄存器的值
        pop ebx                         ; 恢复主调程序的 EBX 寄存器的值
        ret                             ; 返回主调程序

;---------------------------------------------------------------------
; PrintLine:将 DumpLin 显示到标准输出
; 更新日期：4/15/2009
; 输入参数：没有。
; 返回值：没有。
; 修改：没有。
; 调用：内核 sys_write
; 描述：将十六进制转储行 DumpLin 显示到标准输出。
;使用 INT 80h sys_write. 所有寄存器都被保存起来。

PrintLine:
        Pushad                      ; 保存主调程序的所有通用寄存器
        mov eax,4                   ; 指定 sys_write 调用
        mov ebx,1                   ; 指定文件描述符 1：标准输出
        mov ecx,DumpLin             ; 传递字符串行的偏移量
        mov edx,FULLLEN             ; 传递字符串行的大小
        int 80h                     ; 进行内核调用显示字符串行
        popad                       ; 恢复主调程序的所有通用寄存器
        ret                         ; 返回主调程序

;---------------------------------------------------------------------
; LoadBuff: 通过 INT 80h sys_read 将一个缓冲区从标准输入装满数据。
; 更新日期：4/15/2009
; 输入参数：没有。
; 返回值：从 EBP 中读入的字节数
; 修改：ECX, EBP, Buff
; 调用：内核 sys_write
; 描述：使用 INT 80h sys_read 从标准输入中加载满满一缓冲区数据
; 并将其放入 Buff。
; 因为我们开始了一个新的装满数据的缓冲区，所以缓冲区偏移量计数器 ECX 被设置为零。
; 主调程序必须测试 EBP 中的值：
; 如果在返回时 EBP 中包含零，表明在标准输入中遇到了 EOF(文件结尾)
```

```
; 如果在返回时 EBP 中的值比零小，表明发生了某种类型的错误。

LoadBuff:
        push eax                ; 保存主调程序的 EAX 寄存器
        push ebx                ; 保存主调程序的 EBX 寄存器
        push edx                ; 保存主调程序的 EDX 寄存器
        mov eax,3               ; 指定 sys_read call
        mov ebx,0               ; 指定文件描述符 0：标准输入。
        mov ecx,Buff            ; 传递要被读入数据的缓冲区的偏移地址
        mov edx,BUFFLEN         ; 传递一次要读入的字节数
        int 80h                 ; 调用 sys_read 来填满缓冲区
        mov ebp,eax             ; 保存从文件中读入的字节数，以备后用。
        xor ecx,ecx             ; 清除缓冲区指针 ECX，将其设置为 0
        pop edx                 ; 恢复主调用程序的 EDX 寄存器
        pop ebx                 ; 恢复主调用程序的 EBX 寄存器
        pop eax                 ; 恢复主调用程序的 EAX 寄存器
        ret                     ; 返回主调用程序

        GLOBAL _start

; ------------------------------------------------------------------
;主程序从这里开始
;-------------------------------------------------------------------
_start:
       nop               ; 为 GDB 输入无操作指令
       nop

; 在循环扫描开始前需要做的所有初始化工作都从这里开始：
       xor esi,esi       ; 将整个字节计数器清零
       call LoadBuff     ; 从缓冲区读入第一个缓冲区的数据
       cmp ebp,0         ; 如果 ebp=0，sys_read 到达了标准输入的 EOF(文件结尾)
       jbe Exit

; 审查该缓冲区，并将这些二进制字节转换为十六进制数字值：
Scan:
       xor eax,eax                 ; 将 EAX 寄存器清零
       mov al,byte[Buff+ecx]       ; 将一个字节从缓冲区读出到 AL 中
       mov edx,esi                 ; 复制整个计数器的值到 EDX 寄存器中
       and edx,0000000Fh           ; 屏蔽掉字符计数器的最低四位
       call DumpChar               ; 调用字符插入过程

; 将缓冲区的指针指向下一个字符，并查看一下缓冲区的工作是否已经完成：
```

```
        inc esi              ; 递增整个已处理的字符的计数值
        inc ecx              ; 递增缓冲区的指针值
        cmp ecx,ebp          ; 与缓冲区中的字符数相比较
        jb .modTest          ; 如果我们已经处理完了缓冲区中的所有字符......
        call LoadBuff        ; ......再次充满该缓冲区
        cmp ebp,0            ; 如果 ebp=0, sys_read 到达了标准输入的结尾
        jbe Done             ; 如果到达了 EOF，我们的任务就完成了

; 判断一下我们是否到达了一个包含 16 个值的块的结尾，并且需要显示一行：
.modTest:
      test esi,0000000Fh   ; 测试计数器中最低的四位是否为零。
      jnz Scan             ; 如果计数器不是 16 的余数，继续循环
      call PrintLine       ; ......否则打印那一行
      call ClearLine       ; 清除十六进制转储行，将其全部设置为 0
      jmp Scan             ; 继续扫描缓冲区

; 全部完成！让我们结束这个“晚会”：
Done:
       call PrintLine       ; 打印“剩余”行
Exit:  mov eax,1            ; 用于退出 Syscall 的代码
       mov ebx,0            ; 将零作为返回码
       int 80H              ; 进行内核调用
```

我承认，这个程序看起来有点吓人。它有 200 多行代码，是到目前为止本书中最重要的也是最大的程序。不过，它所做的事情却很简单。这是来自清单 9-1 的 hexdump1 程序的一个简单扩展。如果你还记得，十六进制转储程序能够读入任意形式(文本、可执行文件、二进制数据，不管什么)的文件，并将其显示在屏幕上(这里为显示到 Linux 控制台上)，这样，每个程序字节都以十六进制的形式给出。清单 9-1 做了其中大部分工作。hexdump2 增加的是另外一个显示列，在该列中，任何可打印的 ASCII 字符(字母、数字、符号)都以它们的本来面貌显示出来，而那些不可打印的字符则用一个占位符字符来表示。这个占位符字符通常是一个 ASCII 句号字符，但是那只是一个习惯，它也可以是任何内容。

可以使用 hexdump2 来显示任何 Linux 文件的十六进制转储，并且采用以下方式来调用它：

```
$./hexdump2 < (filename)
```

I/O 重定向操作符< 采用右侧文件名中指定的文件中的所有数据，并将这些数据输送到标准输入。hexdump2 程序从标准输入中取出数据，并将其以十六进制转储格式打印出来，每行显示 16 个字节，整个文件需要显示多少行，它就显

示多少行。

例如，hexdump2 程序本身的生成文件的十六进制转储如下所示：

```
68 65 78 64 75 6D 70 32 3A 20 68 65 78 64 75 6D |hexdump2: hexdum|
70 32 2E 6F 0A 09 6C 64 20 2D 6F 20 68 65 78 64 |p2.o..ld -o hexd|
75 6D 70 32 20 68 65 78 64 75 6D 70 32 2E 6F 0A |ump2 hexdump2.o.|
68 65 78 64 75 6D 70 32 2E 6F 3A 20 68 65 78 64 |hexdump2.o: hexd|
75 6D 70 32 2E 61 73 6D 0A 09 6E 61 73 6D 20 2D |ump2.asm..nasm -|
66 20 65 6C 66 20 2D 67 20 2D 46 20 73 74 61 62 |f elf -g -F stab|
73 20 68 65 78 64 75 6D 70 32 2E 61 73 6D 0A 00 |s hexdump2.asm..|
```

生成文件是纯文本，因此在转储中并没有很多不可打印字符。不过请注意，tab 和 EOL 这两个不可打印字符一般存在于 Linux 的文本文件之中，它们是可以被清晰地看到的，无论是在左栏的十六进制形式，还是在右侧列以句号的形式。这是有用的，因为当该文件作为纯文本显示在控制台上时，tab 字符和 EOL 字符是不可见的。(它们对文字排版的影响可以看得到，但是你无法看到这些字符本身。)

拥有一个文件的十六进制转储能够精确地向你显示在该文件的哪些地方有 tab 字符和 EOL 字符出现，以及在任一特定位置有多少个它们出现。

鉴于 hexdump2 比较复杂，在我们深入研究过程的内部机制原理之前，通过伪代码来告诉你该程序的工作原理可能会比较有用，下面就是从某一(较高的)高度来看的该程序的工作原理：

只要从 `stdin` 中可以得到数据，就执行下列操作：

从标准输入中读入数据。
将数据字节转换为合适的十六进制/ ASCII 显示形式。
将格式化数据字节插入到一个 16 字节的十六进制转储行中。
每 16 个字节，显示一下该十六进制转储行。

当你知道大概想要程序做哪些事情，但是对于如何准确地完成这些事情仍然有些模糊的时候，这是一个早期伪代码迭代的很好示例。它能够让你对于下面的更为详细的伪代码有个抢先理解：

将总字节数(ESI)和偏移计数器(ECX)清零。
调用 LoadBuff 以从标准输入中取出首批数据来填充缓冲区。
测试从标准输入读入到缓冲区中的字节数。
如果字节数为 0，则该文件为空文件，跳转到 Exit 标号处。

Scan: 从缓冲区中取出一个字节并将其放入 AL 中。
推断出该字节在十六进制转储字符串行中的位置。
调用 DumpChar 将该字节插入到该字符串行中。
递增总计数器和缓冲区偏移计数器。
测试一下，看看是否已经处理完了缓冲区中的最后一个字节：
如果是这样，调用 LoadBuff 以从标准输入中读入数据来填充缓冲区。
测试一下从标准输入读入缓冲区中的字节数。
如果字节数为 0，我们到了文件的结尾 EOF；跳转到标号 Exit 处。
测试一下，看看我们是否已经将 16 个字节插入到十六进制转储行中。
如果是这样，调用 PrintLine 来显示该十六进制转储行。
回到标号 Scan 处，继续循环。
Exit: 为每个 Linux 请求正常关闭该程序。

与第 8 章中展示的伪代码示例不同的是，这里有对于过程的明确引用。我想它们几乎能够根据上下文不言自明，而这些上下文就是衡量一个过程编写得好坏的标志。例如，“Call LoadBuff”的意思是“执行一个能够实现加载缓冲区的过程”。这就是 LoadBuff 过程所做的事情，也是 LoadBuff 所做的全部事情。你不必面对 LoadBuff 完成其工作的所有细节。这使得我们能够更加容易地从整体上把握整个程序的逻辑流。

再次通读一下清单 10-1 中的代码，看看你能否理解将上述伪代码与实际的机器指令相关联的原理。一旦你已经掌握了，我们就可以开始对过程展开更加深入的谈论了。

10.2 调用和返回

在主程序块 hexdump2 的开始附近，有一条本书前面部分没有用过的机器指令：

```
call LoadBuff
```

标号 LoadBuff 指的是一个过程。正如你可能已经了解到的那样(特别是当你已经使用过一种较旧的语言编程，如 BASIC 或 FORTRAN 语言)，Call LoadBuff 简单地告诉 CPU 去执行一个名为 LoadBuff 的过程，然后当 LoadBuff 完成运行时再返回来。LoadBuff 之前已经在清单 10-1 中进行了定义，但是为了使以下的讨论更加清晰，我把它复制在下面。

LoadBuff 是一个很好的关于过程的入门级示例，因为它在逻辑上相当的直接，并且它使用的是已经讨论过的指令和概念。像一般的汇编语言程序一样，过程(如 LoadBuff)从其顶部开始执行，顺序运行内部的指令，在某一时刻结束。结束不一定必须在指令序列的最底部发生，但是一个过程的“结束”之处始终是该过程返回到主程序调用它的地方。这个地方就是我们看到的 CALL 指令“改变自我”的地方：RET(返回的缩写)。

```
LoadBuff:
    push eax                ; 保存主调程序的 EAX 寄存器。
    push ebx                ; 保存主调程序的 EBX 寄存器。
    push edx                ; 保存主调程序的 EDX 寄存器。
    mov eax,3               ; 指定 sys_read 调用。
    mov ebx,0               ; 指定文件描述符 0:标准输入
    mov ecx,Buff            ; 传递要读入数据的缓冲区的偏移量
    mov edx,BUFFLEN         ; 传递要在一次循环中读入的字节的个数
    int 80h                 ; 调用 sys_read 来填充缓冲区
    mov ebp,eax             ; 保存从文件中读入的字节数，以备后用
    xor ecx,ecx             ; 将缓冲区指针 ECX 清零
    pop edx                 ; 恢复主调用程序的 EDX 寄存器
    pop ebx                 ; 恢复主调用程序的 EBX 寄存器
    pop eax                 ; 恢复主调用程序的 EAX 寄存器
    ret                     ; 返回主调用程序
```

在比较简单的示例中(如 LoadBuff)，RET 一般位于过程的指令序列的最后。然而，RET 可以出现在过程中的任何地方，而且某些情况下你可能会发现，在一个过程中存在不止一个 RET。到底哪个 RET 实际上将执行带回了主调程序，依赖于过程所做的事情是什么，它遇到的情况下是什么，但是这并不重要。每一个 RET 都是一个返回到调用该过程的代码的“出口点”，而且(更重要的是)一个过程内的所有 RET 指令都将执行带回到一个同样的位置：调用该过程的 CALL 指令后面的那条指令。

过程的结构要点如下：

- 过程必须首先从一个标号开始，标号是(你应该记得)一个后面跟着一个冒号的标识符。
- 过程内部必须至少有一条 RET 指令。
- 过程可以有一个以上的 RET 指令。执行必须通过 RET 指令从过程中返回，而一个过程的出口可以有不止一个，到底采用哪个出口取决于该程序的执行流程。但是通过条件转移指令，你可以从任何满足过程逻辑要

求的地方退出。

- 一个过程可以通过使用 CALL 指令来调用另外一个过程。(稍后会有更多关于这方面的介绍。)

CALL 和 RET 操作所采用的方法听起来可能非常熟悉：CALL 首先将位于它本身后面的下一条指令的地址压入堆栈。然后 CALL 将执行转移到由命名过程的标号所代表的地址处——在本例中为 LoadBuff。过程中所包含的指令将得到执行。最后，过程通过指令 RET 而终止。RET 指令弹出堆栈顶部的地址，并将执行流程转移到该地址。因为压入堆栈的地址是 CALL 指令之后的第一条指令的地址，所以执行将继续进行，仿佛 CALL 根本没有改变过指令执行的流程一样(见图 10-1)。

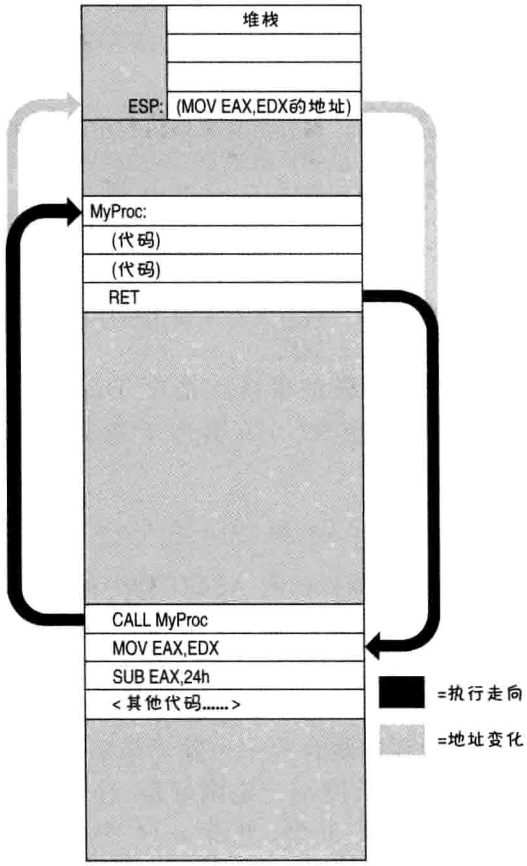


图 10-1 调用过程和从过程中返回

这可能让你想起了第 8 章中讲解 INT 指令相关知识时介绍过的软件中断的工作原理。主要的不同是，主调程序知道它希望调用的代码的确切地址。除此之外，二者的执行过程非常相近。但是请注意，RET 和 IRET 是不能互换的。CALL 指令与 RET 指令一起工作，就像 INT 指令与 IRET 指令一起工作一样。千万不要让这些返回指令搞糊涂了！

10.2.1 调用中的调用

在过程内部，可以做任何主程序本身能够做的事情，包括从一个过程内部调用其他过程以及通过 INT 80h 调用请求 Linux 内核服务。

在 hexdump2 中有一个简单的例子：在 ClearLine 过程中调用了 DumpChar 过程，用来实现将该十六进制转储行变量 DumpLin 清零的功能：

```
ClearLine:
    pushad                  ; 保存所有主调用程序的通用寄存器。
    mov edx,15              ; 我们将执行 16 次插入操作，从 0 开始计数
.poke: mov eax,0            ; 告诉 DumpChar 插入一个 '0'
    call DumpChar           ; 将'0'插入到十六进制转储字符串中
    sub edx,1               ; DEC 并不影响 CF!
    jae .poke               ; 如果 EDX >= 0，继续回去循环
    popad                   ; 恢复所有主调用程序的通用寄存器。
    Ret                     ; 返回
```

大致说来，ClearLine 过程所做的事情就是对 DumpChar 过程的一种使用特例，我将在稍后对其进行详细解释。当填充了数据并显示到控制台上时，DumpLin 变量的内容如下所示：

```
75 6D 70 32 2E 61 73 6D 0A 09 6E 61 73 6D 20 2D |ump2.asm..nasm -|
```

每个两字符的十六进制值，以及右侧 ASCII 列中的每个 ASCII 字符，都通过简单的调用 DumpChar 过程插入进来。要想“填充”DumpLin 变量，需要调用 16 次 DumpChar 过程。然后就可以显示它了。在 DumpLin 显示到控制台上以后，hexdump2 继续循环，并开始再次填充 DumpLin。每调用 DumpChar 16 次之后，hexdump2 都将 DumpLin 显示到控制台上……除了最后一次之外。

被显示到控制台上的文件的长度不一定刚好是 16 的整数倍，所以 DumpLin 最后显示的可能是一个包含 2 个、3 个、9 个、11 个，或者任意小于 16 个字符的、不完整的行，我将其称为“余数”。当显示一个局部行时，转储行中的最后

几个字节可能是以前发送到控制台上的 DumpLin 的显示数据。为了避免这种情况，在每次显示之后应该将 DumpLin 清零。这就是 ClearLine 过程所做的事情。经过对 ClearLine 过程的一次调用之后，DumpLin 的内容如下：

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

ClearLine 完成一项简单而明显的事情：它调用 DumpChar 过程 16 次，每次传递给 DumpChar 一个 EAX 中的值——0。DumpChar 不仅插入一个十六进制值 00 的 ASCII 等值，还插入一个 ASCII 句号，用来代表 ASCII 列中的每一个 0 值。(0 不是一个可显示的 ASCII 字符。)

10.2.2　意外递归的危险

在一个过程内部调用另外一个过程时，需要至少注意一件事情：堆栈空间。记住，每个过程调用都将一个 32 位的返回地址压入堆栈。只有当该过程的 RET 指令得到执行时，这个返回地址才会被从堆栈中移除。如果你在一个过程返回之前执行了另外一条 CALL 指令，那么第二个 CALL 指令将另一个返回地址压入堆栈。如果你继续在过程的内部调用过程，那么对于每一个调用都会有一个返回地址堆放到堆栈上，直到你开始从所有这些嵌套的过程中返回。

这曾经是一个在 DOS 下面临的真正问题，当内存稀少时，程序可能只分配几百个字节的内存给堆栈，有时甚至少于这个数字。每一个压入堆栈的地址都会使堆栈朝着该程序的数据段(.data)和代码(.text)段的方向向下增长。调用“过深”可能会使堆栈与数据或代码相撞，进而造成程序的崩溃，而这个崩溃通常会使与它一起工作的 DOS 也陷入崩溃。在 Linux 下，你拥有一个很大的内存，要想陷入麻烦，你将不得不将过程嵌套几百万层那么深，那将的确是一个规模可观的程序。

但是，你仍然能够通过滥用一种名为递归的高级编程技术而使自己陷入麻烦。在递归中，一个过程通过调用它本身来完成其工作。这在初学者看来似乎有些怪诞，但是对于某些特定的程序逻辑而言，它是一种合法的表达方式，而且颇受推崇。当然，递归知道何时停止。对于每一个对自身的调用，递归过程必须最终执行一个 RET。即使递归过程调用本身几十次或几百次，只要 CALL 指令平衡了 RET 指令，就不会发生任何糟糕的事情。

如果你编写的递归过程不正确，而且决定何时使用那条至关重要的 RET 指令的逻辑编码有误，问题就来了。何时返回通常由一个条件转移指令来控制。如

果没有遵循该条件转移指令的使用规则，或者对其标志规范理解有误，那么该过程将永远不会返回，而是继续一遍，一遍，又一遍地自己调用自己。在现代 PC 中，一个汇编语言过程可以在一秒以内甚至更少的时间里调用自己一百万次。此时，堆栈会与代码相撞，Linux 将提交一个分段错误。

我不打算在本书中讲解递归，我提到它只是因为你有可能意外地使用递归。为了符合目前的例子，假设你是在深夜对 ClearLine 进行编码，并且在 ClearLine 调用 DumpChar 的地方，你稀里糊涂地写下了 CALL ClearLine。不要摇头：我曾经不止一次做过这类事情，迟早你也会遇到这种情况。ClearLine 没有被设计为递归程序，因此它将进入一个无限循环，不停调用它本身直到耗尽了堆栈空间，并触发一个分段错误。

当看到 Linux 提交给你一个分段错误时，请将“意外递归”添加到你要寻找的 bug 列表中。它属于我所说的“罕见但不可避免”的 bug。

10.2.3　一个需要提防的标志规范 Bug

在谈论漏洞时，ClearLine 过程未免显得太过简单，它所完成的工作也相当简单。但是，它仍然提供了一个与标志相关的、初学者非常易犯的、有关漏洞(bug)的教学案例。请看下面的 ClearLine 编码的另一种替代方法：

```
ClearLine:
   pushad              ; 保存主调程序的所有通用寄存器
   mov edx,15          ; 我们将进行 16 次插入工作
.Poke: mov eax,0       ; 告诉 DumpChar 插入一个 '0'
   call DumpChar       ; 将'0'插入到十六进制转储字符串中
   dec edx             ; 这将不会工作!!!!!
   jae .Poke           ; 如果 EDX >= 0，则回去继续循环
   popad               ; 恢复所有主调程序的通用寄存器
   ret                 ; 返回
```

这能够正确工作吗？如果你认为能的话，请再仔细想一想。不错，我们从 15 到 0 倒着计数，从而执行一个简单的循环 16 次。不错，当我们倒计数直至零时，DEC 指令在循环中使用了很多次。但是，这个循环稍有不同，因为当 EDX 寄存器中的计数值为零时我们需要做一些工作，然后再多执行一次递减操作。这里显示的条件转移指令是 JAE，高于或等于则跳转。当 EDX 中的计数值低于零时，它必须跳回到 Poke。DEC 将计数值递减到零，然后低于零也是可以的，所以很奇怪，为什么 JAE 在 DEC 指令后面不执行跳转呢？这个感觉是正确的。

然而，这个标志规范是错误的，如果查阅本书的指令参考资料，你会看到当 CF=0 时它发生跳转。CPU 并不了解 JAE 中的“感觉”。它不是一个大脑，它只是一堆非常小的、非常干净的沙子。它的所有理解是：当 CF=0 时 JAE 指令发生跳转。如果查找 DEC 指令，并且仔细审查标志列表，你会看到 DEC 根本不影响进位标志 CF，并且 CF 标志是在 DEC 指令决定是否跳转之前被检查的。

这就是为什么我们使用 SUB 指令来递减本例中的计数器寄存器的原因，因为 SUB 指令的确会影响 CF，并且允许 JAE 指令正确地工作。不存在速度问题；SUB 指令的执行速度与 DEC 指令的执行速度完全相同。这里的教训是：你需要了解条件转移指令如何解释各种标志。转移给人的感觉有可能具有一定的欺骗性。这就是标志规范的重要性。

10.2.4 过程及其所需的数据

程序通过操作数据使其工作得到执行：在缓冲区中的数据、在已命名变量中的数据以及在寄存器中的数据。过程通常被创建用来对某种特定类型的数据执行某种单一的操作。调用这样的过程的程序将它们作为数据“绞肉机”：一种类型的数据进去，转换后的另一种数据出来。

此外，我们通常需要提供一些数据给过程，以控制或引导其执行的工作。例如，一个过程可能需要一个计数值来知道它到底需要执行某个操作多少次，或者出于某种原因它可能需要一个位掩码来应用到某些数据上，并且每次使用的位掩码可能有所不同。

当编写过程时，你需要决定该过程需要什么数据来完成它的工作，以及如何将这些数据提供给该过程。在汇编(和在大多数非另类的编程语言) 工作中，根据访问方法的不同，一般分为两种类型的数据：全局数据和局部数据。

在纯粹的汇编工作中，全局数据是非常常见的，尤其是在那些类似本书示例的小巧程序中。全局数据可被程序中任意位置的代码访问。全局数据项在程序的.data 或者.bss 段中进行定义。CPU 寄存器也可作为全局数据的容器，因为寄存器是 CPU 的一部分，所以可从程序中的任何地方进行访问。

当你将一个程序分为解为一个主程序和很多组名为库文件的过程时，全局数据的概念变得越来越复杂，本章稍后部分将会对其进行讲解。

对于简单的程序而言，向一个过程传递数据的显而易见的方式往往是最好的方式：将数据放入一个或多个寄存器中，然后再调用该过程。我们已经从使用 INT 80h 指令调用 Linux 内核服务这个例子中看到了这种机制的工作情况。将服

务编号放入 EAX 寄存器中，将文件描述符放入 EBX 寄存器中，将字符串的地址放入 ECX 寄存器中，将字符串的长度放入 EDX 寄存器中，然后通过 INT 80h 进行内核服务调用。

上述规律也适用于一般过程。你编写一个过程，假设当该过程开始运行时，它所需要的值存放在某一特定寄存器中。那么在开始调用该过程之前，你必须要确保调用这个过程的代码能够将正确的值放到正确的地方，除此之外确实没有什么更为复杂的内容。

我们可以通过带有括号的内存寻址表达式来访问过程中的表、缓冲区和其他的已命名数据项，就像它们来自程序的任何其他部分一样。

10.2.5 保存主调程序的寄存器

一旦开始使用汇编语言编写比较重要的程序，你会发现自己永远无法拥有足够的寄存器，并且(不像高级语言如 C 和 Pascal)当需要时，不能创造出更多的寄存器来。寄存器必须要小心使用，你会发现，在任何一个比较重要的程序中，一般情况下所有的寄存器在所有的时间都处于使用状态。

从主程序内部(或从另外一个过程的内部)进入一个过程能够带来一个特定且微妙的问题。你可以从任何地方调用过程，这意味着在该过程被调用时，你不会总能知道哪些寄存器已经被使用。正如上一节中讲解的一样，寄存器是主调程序向过程传递数据的主要方式。过程需要寄存器来进行工作，其他的过程和主调程序同样也需要。没有过程可以假设在它被调用期间，EAX 或 EBP 或任何其他寄存器总是处于“空闲”状态。

这就是为什么编写的比较好的过程在它们开始加载新的数据到寄存器或者对寄存器中的数据进行修改之前，始终保存任何它们将要修改的寄存器的值的原因。如果一个过程只检查某个寄存器的值(但是不改变它)，那么就没有必要保存它了。例如，一个程序可以假设某个寄存器包含一个提供到某个表的索引的计数器值，那么，只要不对该寄存器的值进行修改，它就可以自由地使用那个寄存器。然而，只要过程修改某个寄存器的值 (除非主调程序明确期望返回一个值到该寄存器中)，它就应该保存这个寄存器的值，然后在执行 RET 指令返回主调程序之前再将其恢复为原来的值。

保存主调程序的寄存器的值可以通过 PUSH 指令来完成：

```
push ebx
push esi
```

```
push edi
```

每个 PUSH 指令将一个 32 位寄存器的值压入堆栈。那些值将继续安全地存放于堆栈上，直到返回主调程序之前被弹回到原来的寄存器中：

```
pop edi
pop esi
pop ebx
ret
```

这里有一个绝对关键的细节，一个能够导致很多非常特殊的程序漏洞的细节：主调程序的值从堆栈中弹出的顺序必须与它们被压入堆栈的顺序相反。换句话说，如果你最先压入 EBX，其次是 ESI，再次是 EDI，那么你必须按照先弹出 EDI，接着是 ESI，最后是 EBX 的顺序将其从堆栈中弹出。CPU 将按照任何你指定的顺序，乖乖地将它们弹出到任何寄存器中，但是如果顺序不对，你将会从根本上改变主调程序的寄存器值，而不是保存它们。本应存在于 EBX 中的数据现在可能已经到了 EDX 中，主调程序的逻辑可能会发狂了。

在起初于第 8 章中讲解堆栈时，我已经展示了这种情况如何发生，但是在那个时候，你可能还没有充分理解。快速翻回到图 8-3，看看最右侧列发生了什么。CX 的值被压入堆栈中，但是下一条指令是 POP DX。曾经存在于 CX 中的数据现在到了 DX 里。如果这是你想要的效果，当然很好——有时这可能是解决特定问题的最佳方式。但是，如果将寄存器的值压入堆栈的目的是为了保护它们，那么压栈和出栈的顺序至关重要。

在某些情况下，过程会使用所有通用寄存器中的大部分，有一对指令能够一口气将所有的通用寄存器压入和弹出堆栈。回顾一下前面讲过的 ClearLine 程序。该过程中的第一条指令是 PUSHAD，而且在 RET 指令之前的最后一条指令是 POPAD。PUSHAD 将所有的通用寄存器压入堆栈，包括堆栈指针寄存器 ESP。POPAD 按照正确的顺序将所有那些压入堆栈的寄存器的值弹回到正确的寄存器中。(ESP 的值是一种特例，尽管它的值被压入了堆栈，但是当执行时，PUSHAD 摒弃了从堆栈中弹出的 ESP 的副本。)

我在 ClearLine 中使用 PUSHAD 和 POPAD 是出于特殊的原因：ClearLine 没有返回值给主调程序。它完成一项简单的工作，它的所有行为都以 DumpLin 变量为焦点。因为不需要寄存器向主调程序发回任何内容，所以我选择保留一切，甚至保留不受 ClearLine 影响的寄存器。

这难道不是浪费吗？不一定。不错，将一个寄存器的值压入堆栈的确需要时

间，但是要记住：在任何权衡执行某条指令是否比执行其他指令需要更多时间的时候，你都必须要考虑那条指令将被执行多少次。如果指令是在一个顺序执行十万次、几千万次的紧凑循环中，那么指令速度非常重要。反之，如果一条指令只是在一个程序的运行过程中执行几次，那么它的速度充其量只是一个次要考虑因素。在 PrintLine 过程中，对于每 16 个 hexdump2 处理的字节，ClearLine 只执行一次；即使使用了 PUSHAD 和 POPAD，其执行时间也是只是它前面的 INT 80h 调用 Linux 内核服务所需时间的几分之一。

PrintLine 采用同样的方式：它的执行所耗费的时间被“淹没”在由 INT 80h 指令所做的 sys_write 调用之中，因此使用 PUSHAD 和 POPAD 绝不会影响整个程序的可感知速度。

当然，如果主调程序希望一个过程通过寄存器传回一个值，那么该过程就不能使用 PUSHAD 和 POPAD 了。在这种情况下，你需要鉴别一下哪些寄存器必须要为主调程序保留，哪些寄存器必须用于在返回时带出执行结果。作为一个很好的例子，考虑一下 LoadBuff 过程。LoadBuff 保留了 EAX、EBX 和 EDX 3 个主调程序的寄存器的值。但是，对另外两个寄存器(ECX 和 EBP)它所做的事情是：修改它们的值，而不是保存它们的原值。

为什么呢？因为 ECX 寄存器中包含着一个“全局”值：文件缓冲区变量 buff 中的下一个即将被处理的字符的位置。当满满一缓冲区数据全部处理完毕时，必须从标准输入中加载新的数据，我们通过调用 LoadBuff 过程来实现。当重新填充缓冲区时，缓冲区的计数器必须被重置为 0，以便处理过程可以重新开始，从头处理那些新数据。LoadBuff 做的就是这件事情，这个复位后的 ECX 被传回主调程序。

EBP 也有一个使命：它带回由 INT 80h 调用 sys_read 加载到缓冲区中的字节数。要想调用 sys_read，必须知道位于程序开始处附近的 BUFFLEN 等值所指定的字节数。但是，由于很少有文件长度刚好为 BUFFLEN 的整数倍，所以，最后一批从标准输入中带来的字节数可能比 BUFFLEN 少。这个值也被认为是全局的，主程序根据它来判断什么时候当前缓冲区已经处理完毕。

请注意，主调程序也能保存自己的寄存器，有时它就是这样做的。例如，考虑下面的指令序列：

```
push ebx
push edx
call CalcSpace
pop edx
```

```
pop ebx
```

从功能上来讲，这与从过程内部保存寄存器确实没有什么不同。不过，在程序内部可能不止一次调用 CalcSpace 过程。可能有几十次，或几百次调用，每个这样的调用都需要五条指令，而不是只有一条。如果寄存器的保存工作是在过程内部完成，那么这个保存仅需要 5 条指令，无论在代码中有多少个位置调用该过程。

关于哪些寄存器需要保存，没有严格且快速的规则。你需要知道寄存器在给定程序的任一位置如何被使用，并据此编程。(当你设计自己的程序时，对寄存器的使用好好记笔记非常重要。) 在避免漏洞这方面，我能提供的唯一忠告就是谨慎：保存任何你知道的既不在全局范围内使用，也不用来向主调程序返回任何值的寄存器。与由于寄存器冲突而造成的漏洞恶化相比，保存寄存器所需的时间是微不足道的。

10.2.6　局部数据

不同于全局数据，局部数据是只可以在某一特定过程内部，或在某些情况下在某一库文件内部访问(也叫作“可见”)的数据。(同样，让我们暂时推迟有关代码库的讨论。)如果一个过程拥有局部数据，那么当该过程被调用时，这些数据几乎总是存放在堆栈上。

PUSH 指令把数据存放到堆栈上。当部分代码通过 CALL 指令调用过程时，通过在使用 CALL 指令之前一次或者多次使用 PUSH 指令能够将数据传递给该过程，然后这个过程就可以访问这些被压入堆栈的数据项了。但是，这里有一句警告：该过程不能仅仅从堆栈中弹出这些数据项到寄存器，因为返回地址挡在了路上。

请记住，CALL 指令做的第一件事情是将下一条机器指令的地址压入堆栈。当过程得到控制时，其返回地址位于堆栈的顶部(我们通常称之为 TOS)，为实现返回的那条必不可少的 RET 指令做好准备。任何在 CALL 指令之前被主调程序压入堆栈的数据项都高于返回地址。这些数据项仍然可以通过使用普通的存储器寻址和堆栈指针 ESP 来进行访问。但是在没有将返回地址弹出和重新压入堆栈之前，你不能使用 POP 指令来获取它们。

这的确能够工作，我曾经用过一两次，但是它的执行速度很慢，而且一旦你了解了堆栈帧以及如何在一个堆栈帧中进行内存寻址的本质，根本没必要使用

它。我将在本书的后面部分讲解堆栈帧的概念，因为一旦你开始调用使用 C 或者其他高级语言编写的库过程，它绝对是至关重要的。现在，只需简单理解：全局数据在程序的.data 段和.bss 段进行定义；而局部数据则存放在堆栈上，只能在某一特定过程的特定调用中“局部”使用。出于安全使用考虑，局部数据需要一些特殊的照顾和纪律，其中的原因将在后面给出。

10.2.7 更多的表格技巧

hexdump2 程序工作起来与来自清单 9-1 的 hexdump1 程序非常相像，但是在它的“黑盒子”里面，存在着更多的技巧。值得一提的是十六进制转储行变量——DumpLin 的定义：

```
DumpLin: db " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "
DUMPLEN EQU $-DumpLin
ASCLin: db "|................|",10
ASCLEN EQU $-ASCLin
FULLLEN EQU $-DumpLin
```

我们在这里拥有一个由两个部分组成的变量声明。每个部分都可单独使用，或者(如通常所做的那样) 同时使用。DumpLin 的第一部分是包含 16 个十六进制数字的字符串。它的长度由 DUMPLEN 等值来定义。(请注意，我的个人习惯是把等值的名字大写。等值与变量并不属于同一种类，我发现它能够使程序更加易读，将等值装饰一下，这样它们就可以与变量一目了然地区分了。这不是 NASM 中的要求；可以使用小写字母或者大小写字母的混合体来命名一个等值，随你选择。)

DumpLin 的第二部分是 ASCII 列，它拥有自己的标号：ASCLin。一个只需要 ASCII 列的程序可以单独使用 ASCLin 变量以及与其相关的长度等值——ASCLEN。由于 DumpLin 的两个部分在内存中是相邻的，所以引用 DumpLin 允许你同时将这两个部分作为一个单元来引用，例如，你想通过 INT 80h 发送一行数据到标准输出。在本例中，计算整行长度的等值是 FULLLEN。

给这两个部分取不同的名字是有益的，因为从这两个部分读出和写入数据的方式并非在所有情况下都相同。请看下面的 DumpChar 过程：

```
DumpChar:
    push ebx                ; 保存主调程序的 EBX 寄存器。
    push edi                ; 保存主调程序的 EDI 寄存器。
```

```
; 首先，我们将输入字符插入到转储行的 ASCII 部分
    mov bl,byte [DotXlat+eax]      ; 将不可打印字符翻译成'.'
    mov byte [ASCLin+edx+1],bl     ; 写入 ASCII 部分
; 下一步我们插入输入字符的十六进制等值到转储行的十六进制部分:
    mov ebx,eax                  ; 将输入的字符另存一份
    lea edi,[edx*2+edx]            ; 计算字符串行中的偏移量(ECX*3)
; 查找低半个字节字符，并将其插入到字符串中:
    and eax,0000000Fh              ; 屏蔽掉所有除了低半字节之外的位
    mov al,byte [HexDigits+eax]    ; 查找与该半字节相等的字符
mov byte [DumpLin+edi+2],al        ; 将与该半字节相等的字符写入字符串行中
; 查找高半个字节字符，并将其插入到字符串中:
    and ebx,000000F0h              ; 屏蔽掉除第二低的半字节之外的所有位
    shr ebx,4                      ; 将高四位移到低四位
    mov bl,byte [HexDigits+ebx]    ; 查找与该半字节相等的字符
    mov byte [DumpLin+edi+1],bl    ; 将与该半字节相等的字符写入字符串行中
; 任务完成! 让我们“打道回府”:
    pop edi                        ; 恢复主调程序的 EDI 寄存器的值
    pop ebx                        ; 恢复主调程序的 EBX 寄存器的值
ret                                ; 返回主调程序
```

写入 ASCII 列非常简单，因为 ASCII 码列中的每个字符在内存中都是单个字节，而任何一个 ASCLin 中的位置的有效地址非常容易计算:

```
mov byte [ASCLin+edx+1],bl       ; 写入 ASCII 部分
```

但是，每一行的十六进制转储部分的位置都由三个字符组成的：一个空格，后面跟着两个十六进制数字。我们把 DumpLin 看作一张表，要想对该表中的每一个特定项进行寻址，在计算其有效地址时都需要进行三倍放大:

```
lea edi,[edx*2+edx]                ; 计算在字符串行中的偏移地址(EDX*3)
```

从数据操作的角度来看，十六进制转储行的两个部分的处理相差很大，只有当发送到标准输出时它们才共同行动。所以，让两部分各自拥有自己的标号是有用的。C 语言中的结构体和 Pascal 中的记录在本质上都以相同的方式进行处理。

DotXlat 表是另外一个字符转换的例子，像所有的此类翻译表一样，表达了在一个文本行中相容地显示 256 个不同的 ASCII 值所需的规则:

- 所有的可打印字符都被翻译成它们本身。
- 所有的不可打印字符(其中包括所有控制字符和从 127 开始以及以上的所有的字符) 被翻译成 ASCII 的句号。(句号的英文是“dot”，因此该表的名字中也包含一个“dot”。)

10.2.8　在过程定义中加入常量数据

现在我们已经习惯性地认为代码是存在于.text 段的，而数据是存在于.data 段或者.bss 段的。在几乎所有情况下，这是一种很好的代码组织方式，但是我们不是必须绝对以这种方式来分开代码和数据。在过程内部使用 NASM 伪指令(包括 DB、DW 和 DD)来定义数据是有可能的。我已经创建了一个有用的过程，该过程显示了如何做到这一点，同时它也是一个有关何时需要这样做的很好示例。

Newlines 过程可以向标准输出发出一定数量的换行符字符，其数量由一个传递给这个子程序的、存放在 EDX 中的值来指定：

```
;-----------------------------------------------------------------------
; Newlines: 发送 1 到 15 个的换行符 到 Linux 控制台
; 最后更新日期: 4/19/2009
; 输入: EDX: #个换行符要发送，范围是从 1 到 15 之间
; 返回值: 没有
; 修改: 没有. 所有主调程序的寄存器都得到保存.
; 调用: 内核 sys_write
; 描述: 通过使用 INT 80h sys_write，将若干个换行符 (0Ah) 发送到标准输出，
; 其中，换行符的个数由 EDX 中的值来指定。
; 本过程演示了将常量数据放到过程定义本身，
; 而不是放到.data 段或者.bss 段。
Newlines:
    Pushad              ; 保存所有主调程序的寄存器
    cmp edx,15          ; 确保主调程序的请求没有超过 15 个
    ja .exit            ; 如果超过，什么都不做就退出，(跳转到标号.exit 处)
    mov ecx,EOLs        ; 将 EOLs 表的地址放入 ECX 寄存器中
    mov eax,4           ; 指定 sys_write
    mov ebx,1           ; 指定 stdout
    int 80h             ; 进行内核调用
.exit popad             ; 恢复所有主调程序的寄存器
    ret                 ; 返回！
EOLs db 10,10,10,10,10,10,10,10,10,10,10,10,10,10,10
```

表 EOLs 包含 15 个 EOL 字符。如果回想一下，你应该还记得，当 EOL 字符被发送到标准输出时，控制台将其解释为一个换行符，控制台的光标位置被下移一行。主调程序将期望的换行符数量传递到 EDX 寄存器中。Newlines 过程首先进行检查以确保主调程序要求提供的换行符不比表中拥有的 EOL 字符更多，然后，通过使用 INT 80h，将 EOLs 表的地址和所需的 EOL 数量传递给一个对

sys_write 的常归调用。从根本上来讲，sys_write 将 EOLs 表的前 ECX 个字符显示到控制台上，控制台将该数据解释为 ECX 个换行符。

将数据直接放在过程内部意味着很容易通过剪切和粘贴将过程定义从一个程序移植到另一个程序中，而不会漏掉 EOL 字符背后那张必不可少的表。因为曾经使用过 EOLs 表的唯一代码就是 Newlines 过程本身，将 EOLs 表放在较集中可见的.data 段并不会带来什么好处。尽管从技术意义来讲，EOLs 表不是局部的(对 Newlines 的调用没有将它放到堆栈上)，但是它“看起来”是局部的，这就避免了.data 段和.bss 段因为一个只在一个单个过程中引用的数据而变得更加杂乱无章。

10.3　局部标号和跳转的长度

随着程序越来越长，越来越复杂，迟早你会由于疏忽而重复使用了某个标号。在本书中我不会提供任何特别长或者特别复杂的程序，所以代码标号彼此冲突的情况不会成为一个实际的问题；但是随着编写的程序越来越重要，你将在一个源代码文件中编写数百行，或者甚至(通过一些实践和毅力)数千行汇编代码。你很快就会发现，代码标号重复使用的问题越来越严峻。你怎么可能永远记住已经在一个 2732 行程序的第 187 行使用过了标号 Scan 呢？

你永远也做不到的，迟早(特别是如果你经常处理缓冲区和表的时候)你会尝试再一次使用标号 Scan。而 NASM 也会将错就错，允许你使用它。

这是一个十分普遍的问题(特别是对于那些明显有用的标号，如 Scan)，NASM 的作者创造了一个特性来处理这个问题：局部标号。局部标号基于这一事实：(子程序名字和主要的段之外的)汇编工作中的几乎所有标号在本质上都是“局部”的，所谓“局部”，我的意思是，它们只能被与它们非常接近(也许只有两三条指令的距离)的转移指令(jump)引用。这种标号通常位于紧密循环中，它不在较远的代码中引用，而且往往只是在一个地方引用。

下面是一个来自 hexdump2.asm 的主体部分的例子：

```
Scan:
    xor eax,eax                 ; 将 EAX 寄存器清零
    mov al,byte[Buff+ecx]       ; 从缓冲区中取出一个字节，放入 AL 中
    mov edx,esi                 ; 复制整个计数值到 EDX 寄存器中
    and edx,0000000Fh           ; 屏蔽掉字符计数器的最低四位
    call DumpChar               ; 调用字符插入过程
;将缓冲区指针指向下一个字符，以判断是否缓冲区已经处理完毕：
    inc esi                     ; 增加整个已处理的字符的计数值
```

```
    inc ecx              ; 增加缓冲区的指针值
    cmp ecx,ebp          ; 与缓冲区中的字符个数相比较
    jb .modTest          ; 如果已经处理完缓冲区中的所有字符
    call LoadBuff        ; 再次充满该缓冲区
    cmp ebp,0            ; 如果 ebp=0, sys_read 到达了标准输入的 EOF(文件结尾)
    jbe Done             ; 如果到达了 EOF，我们的任务就完成了。
; 判断一下我们是否到达一个包含 16 个值的块的结尾，需要显示一行数据：
.modTest:
    test esi,0000000Fh ; 测试计数值的最低四位是否为零。
    jnz Scan             ; 如果计数器不是 16 的余数，继续循环
    call PrintLine       ; 否则打印那一行
    call ClearLine       ; 清除十六进制转储行，将其全部设置为 0
    jmp Scan             ; 继续扫描缓冲区
```

请注意，在标号.modTest 的前面有一个句号。这个句号标志着它是一个局部标号。局部标号对于前面代码中的第一个非局部标号(即第一个没有句号前缀的标号；我们称之为全局标号)而言是局部的。在这个特例中，.modTest 标号所属的全局标号是 Scan。上面显示的程序块儿是程序主体的一部分，该程序扫描输入的文件缓冲区，将输入数据格式化为每行包含 16 个字节的数据行，然后再将这些数据行显示到控制台上。

那么，全局标号通过什么方式“拥有”一个局部标号呢？这是一个与源代码内部可见性有关的问题：局部标号不能被所属全局标号上面的源代码文件部分引用，再次强调一下，局部标号的所属全局标号是指源代码文件中该局部标号上面的第一个标号。

在这种情况下，局部标号.modTest 不能被全局标号 Scan 上面的代码引用。这意味着，在程序中你可以设想在标号 Scan 的“另一边”有第二个局部标号.modTest。只要在两个同名局部标号之间出现全局标号，NASM 区分它们就不会有麻烦。

过程内部也可以存在局部标号。在另一个来自 hexdump2.asm 的例子中，ClearLine 过程里面有一个局部标号.poke。它隶属于标号 ClearLine，因此无法从程序中或者库中任何其他地方的过程中进行引用。(不要忘了，过程名称是全局标号。) 这种隔离在单个过程中不很明显，但是它是正确的，这主要源于这一事实：在(程序中或者库中的)过程的下面，总是要么有另外一个过程，要么在主程序开始处有一个_start 标号。一旦(像我在图 10-2 中做的那样)把它画出来，你会感觉更加明显。

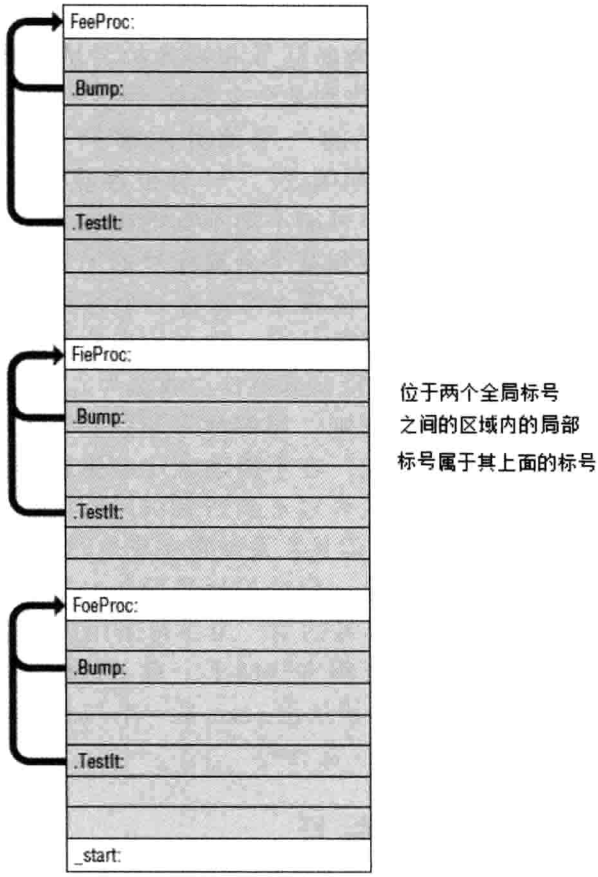


图 10-2　局部标号和拥有它们的全局标号

下面是一些关于局部标号的注意事项:

- 过程内部的局部标号至少对于定义它们的过程而言是局部的。(图 10-2 中已经展示了这一点。)当然，也可以在过程内部定义全局标号，但是这进一步限制了局部标号的可见度。
- 遗憾的是，局部标号不能作为断点供来自 Gdb 调试器的命令行界面访问。我也不很清楚为什么会这样，但是 Gdb 拒绝从命令行设置断点到一个局部标号上。当然，你也能够通过 Insight 的源代码窗口在任意一条包含机器指令的代码行上设置断点，不管有没有标号。一般来说，只有在万不得已的情况下我们才使用 Gdb 的命令行界面；它具有一些特殊的

限制。

- 如果正在编写混合了很多全局标号和局部标号的紧凑代码，你要小心了，不要尝试让 JMP 指令跳到某个全局标号的另外一边的局部标号上。这就是为什么不要让程序的一个部分内部拥有 15 个名为.scan 或者 .loopback 的局部标号的原因之一——你很容易将它们搞混，并且在试图向上跳转五条指令时，你可能不知不觉地跳转到了下面距离它七条指令的某个地方。如果尝试跳到某个全局标号的另外一边的局部标号时，该全局标号在你这边的某个位置上恰好有一个同名标号，NASM 并不会提醒你。这样的漏洞有时很难找到。像使用所有工具一样，我们一定要谨慎使用局部标号，这样才能带来最佳“收益”。
- 这里有一个我使用的经验法则：局部标号和到它们的所有跳转应该发生在一屏代码之内。换句话说，在不滚动程序编辑器的情况下，你应该既可以看到一个局部标号，也可以看到所有引用它的指令。这只是一个帮助你跟踪程序意识的指南，但是我发现它在我自己的工作中非常有用。
- 我还使用一个代码风格约定：将局部标号的句号之后的第一个字符采用小写形式，而将所有全局标号的第一个字符采用大写字符。NASM 并没规定必须这样做，但是我觉得它有助于一目了然地区分局部标号和全局标号。这样，我们就很容易辨认出.poke 是一个局部标号(毕竟，句号是非常小的！)，而 Scan 是一个全局标号。

10.3.1 “强行”访问局部标号

几乎每隔一段时间(也不是很经常)，你会发现需要从某个局部标号所属的全局标号的另一边访问该局部标号。NASM 提供了一种做这件事情的方法，虽然我得承认，我自己从来没有过那种需要。在一个局部标号的范围(它通常可见的那块程序区域)之外对其进行强制访问的关键是在本质上了解 NASM 如何处理局部标号。

局部标号有一个隐含的定义，该定义包括了它所属的全局标号。本节前面讨论过的局部标号.modTest 属于全局标号 Scan。在内部，NASM 将.modtest 看作 Scan.modTest。如果在程序的其他地方有另外一个.modtest 局部标号 (比如，该标号隶属于全局标号 `Calc`)，那么你可以通过在跳转指令中包含它的主人的名字，强行执行一个对它的跳转：

```
jne Calc.modTest
```

从某种意义上来讲，局部标号只是全局标号的“尾巴”；如果需要的话，我们可以通过在某个局部标号的前面加上它的全局拥有者标号来将其转换为一个全局标号，进而实现对它的访问。

再次重申，我从来没有这样做过，我也不认为这是一种很好的做法，但是，如果真有这种需求的话，知道它可以实现总是不错的。

10.3.2 短转移、近转移和远转移

一个你很可能遇到过的、非常奇怪的汇编错误也可能出现在完全正确的程序里面；如果使用 NASM 工作的时间足够长，并且创建的程序足够大，你将会遇到这种情况。这里就有一个：

```
error: short jump is out of range
```

当条件转移指令距离它所引用的标号太远时，将发生这个错误，这里所说的“太远”指的是在内存中很多条指令位置以外。这仅适用于条件转移指令；无条件转移指令 JMP 不会遭遇这种错误。

问题的产生是因为 NASM 为特定的条件转移指令产生二进制操作码的方式有所不同。根据跳转的目标标号的距离，有两种不同类型的条件转移指令。跳转目标在 127 个字节以内的条件转移指令叫作短转移指令。跳转目标的距离超过 127 个字节，但是仍然在当前代码段中的条件转移指令叫作近转移指令。(近转移指令最大可跳转到距离该指令 2GB 的位置，这可能延伸了你对“近”这个词的概念。)

还有第三种转移指令，叫作远转移指令，其中涉及(无论由于何种原因)完全离开当前代码段。在老式的 DOS 实模式世界里，这意味着既要为跳转目标指定一个段地址，又要为其指定一个偏移地址，并且这些指令也不常使用，尽管我曾经用过它们一两次。在 32 位保护模式的世界里，远转移指令的使用更是极为罕见，它涉及各种各样的操作系统的复杂性，我在本书中不会对其进行深入研究。

真正的问题在于短转移指令和近转移指令之间的差异。短条件转移指令生成一个短的(因而比较紧凑的)二进制操作码。该操作码的大小总是两个字节。近转移指令的操作码大小是 4 个或者 6 个字节，取决于各种因素。紧凑的代码意味着快速的代码，并且在大多数情况下，采用短转移指令要(略)比采用近转移指令的速度更快一些。此外，如果大部分时间使用短转移指令，你的可执行文件将变得

更小。

鉴于你将编写的 90%以上的条件转移指令距离程序的目标位置仅相隔几条指令，NASM 默认情况下为短转移指令生成操作码，这一点是有道理的。事实上，NASM 为短转移指令生成操作码，除非你明确告诉它使用近转移指令。近转移指令可以通过使用 NEAR 限定词来指定：

```
jne Scan            ; 短转移指令，可跳转到任意方向的 127 个字节之内的位置
jne near Scan       ; 近转移指令,可跳转到当前代码段的任意位置
```

初学者往往以这种方式遭遇到“短转移越界”错误：你开始一个程序，并且在程序的结尾处放了一个标号，如 Exit：，期待从该程序的几个不同部分跳到这个 Exit 标号。当这个程序是一个新程序，并且体积仍然相当小的时候，它可能会工作得很好。但是，添加到程序中的最终代码迫使程序开始处的条件转移指令与标号 Exit 之间的距离超过了 127 个字节。砰！NASM 提交给你一个“短转移越界”错误。

修复的办法很简单：对于所有 NASM 称之为“越界”的转移指令，在助记符和目标标号之间插入一个 NEAR 限定词。其他的指令则不用动。

10.4　生成外部过程库

你会发现清单 10-1 中的 hexdump2 程序已经将其大部分程序分离到过程中。这是因为出于维持程序的可理解性和可维护性的目的，它应该是这样的。不过，hexdump2.asm 中声明的过程只能被 hexdump2 程序本身使用。如果你编写一个更加强大的程序，不管出于什么原因，需要显示一些数据的十六进制/ ASCII 转储，这些过程可以被再次使用——但是绝不是在它们位于 hexdump2 期间。

答案是将 hexdump2 中的过程从 hexdump2 中完全分离出来，并且放到一个完全独立的源代码文件中。然后这个文件就可以单独编译为一个.o 文件，依次地，这个.o 文件又可以被 Linux 连接器连接到你将来可能编写的其他程序中。像这样的源代码文件就是一个过程库。它里面可能充满了过程，但是却没有主程序部分，也没有用以表明执行开始之处的标号_start:。它包含的所有内容就是过程，并且不能被连接器翻译成它本身的可执行程序。

我曾在第 5 章中描述过独立的汇编过程，并将其形象地显示在图 5-8 和图 5-9 中。一个单个(且相当简单的) 程序可能包括三个或四个独立的.ASM 文件，每一个都被分别编译成一个单独的.O 文件。要想产生最终的可执行文件，Linux 连接

器(Ld)将所有这些.O 文件编织在一起，处理所有从一个文件到另一个文件的引用，最终创建可执行文件。

从编译过程来看，每个单独的.asm 文件都被认为是一个模块，不管它包含一个_start:标号(因而是个程序)，还是简单地包含过程。每个模块都包含代码，可能还有一些数据定义。当所有声明都被正确地完成后，所有模块就可以通过过程调用来互相自由“交谈”了，而且任何过程都可引用被连接器组合在一起的文件中的任何地方的任何数据定义。每个可执行文件只能包含一个 _start:标号，所以在若干个被连接到某一可执行文件的模块中，只有一个模块可能包含_start:标号，因而是一个真正的程序。

它本身的实际情况并不像听起来那么困难。诀窍就是保证所有的声明都是正确的……

10.4.1 全局声明和外部声明

……那需要的技巧比起以前来要少很多。回到糟糕的 DOS 年代，为了使用单独编译的过程库，你必须定义代码段和数据段，并且确保这些段被标识为公有类型(PUBLIC)，你需要不停地做这件事情。在 Linux 中，对于保护模式用户空间程序而言，只有一个段，这个段包含代码、数据和堆栈——即程序拥有的所有内容。大部分以前必须要手动实现的连接，现在由 NASM、连接器和 Linux 加载程序自动实现。现在，创建库文件是小菜一碟，不比创建程序复杂，甚至在某些方面更加容易。

模块化程序设计的核心思想是将地址解析一直推迟到连接时。如果你开始编写自己的汇编程序，也许已经遇到过地址解析问题。它可能在无意中发生：比如你打算在一个程序内部编写一个过程，但是一时冲动先编写了引用那个(尚未编写的)过程标号的代码，NASM 将用一个错误信息来欢迎你：

```
error: symbol 'MyProc' undefined
```

在模块化编程中，经常会调用一些并非真正存在于当前工作的源代码文件中的过程。那么该如何通过汇编的监管机构呢？

答案是声明一个外部过程。顾名思义，所谓“外部”，是指：汇编编译器被告知，将来只能在程序之外的另外一个模块中的某个地方才能找到某一特定标号。一旦得知这个消息，NASM 将爽快地让你通过一个未定义标号。你已经向 NASM 承诺将会稍后提供它，并且 NASM 接受了你的承诺。它将该引用标识为

外部引用，并且保持不会错误地调用那个未定义的标号。

你对 NASM 所做的“承诺”看起来像这样：

```
EXTERN MyProc
```

这里，你已经告诉了汇编编译器，标号 myproc 代表一个过程，而且该过程会在当前模块之外的某处被找到。这就是汇编编译器不弹出错误信息所需知道的一切内容；并且在做完了这件事情之后，汇编编译器的交涉部分就完成了。它在你的程序中留下一个“空的插座”，而外部过程的地址将来会插入进来。我有时将其想象成一个小孔，外部过程将来就挂到这个小孔上。

在 myproc 过程实际存在的其他模块中，仅仅定义该过程是不够的。孔需要一个挂钩。你必须警告汇编编译器，MyProc 将会被从模块外部引用。汇编编译器需要忘掉将要挂到小孔上的挂钩。你可以通过将该过程声明为全局(*global)*过程来忘掉这个挂钩，这意味着程序中的任何其他地方的其他模块都可以自由地引用这个过程。将一个过程声明为全局过程与将其声明为外部过程相比，并不复杂：

```
GLOBAL MyProc
```

总之：一个在定义时被声明为 `GLOBAL` 的过程可被从任何一个标号被声明为 EXTERNAL 的地方进行引用。

小孔和挂钩两个东西都各就各位之后，是谁实际上连接它们呢？连接器在连接过程中做的就是那件事情。在连接时，连接器采用两个由汇编编译器生成的.o 文件，一个来自你的程序，另一个来自包含 myproc 的其他模块，并将这两个文件结合成一个单个的可执行文件。(.o 文件的数量不仅限于两个；你几乎可以在一个单个的程序中使用任意数量的单独编译的外部模块。) 当连接器创建的可执行文件被加载和运行时，该程序可以迅速且干净地调用 `MyProc` 过程，就好像它们都已经在同一个源代码文件中声明了一样。

这个过程以图形的方式来概括，如图 10-3 所示。

过程的工作机制同样也适用于数据，而且它可以在任一方向工作。程序可以将任一已命名变量声明为全局变量，然后，这些变量就可以在那些通过外部指令将相同的变量名声明为外部变量的任一模块中使用了。最后，只要所有的全局声明和外部声明都已被正确处理，过程库们就可以彼此之间共享数据和以任意方式组合了。一个包含声明为全局的(global)过程或者变量的程序或模块常被叫作导出数据项。类似地，一个包含被声明为外部的(external)过程或者变量的程序或模块常被叫作导入数据项。

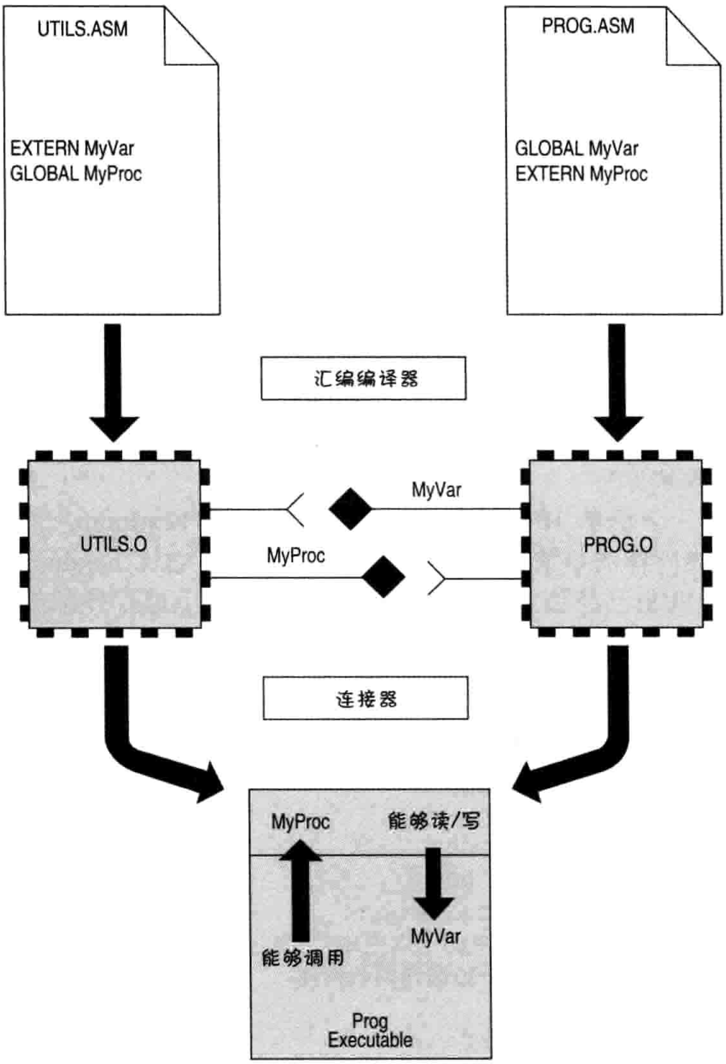


图 10-3 连接全局过程和外部过程

10.4.2 全局过程和外部过程的机制

清单 10-1 中的 hexdump2 程序包含了几个过程。让我们将那些过程从主程序模块中提出来，并根据它们创建单独的汇编库模块，以便观察它们的工作过程。

我已经在前面的几个章节中详细地描述了汇编语言程序的源代码需求。单独

编译的库模块跟程序类似，并且可能拥有所有程序模块可能拥有的 3 个段(.text 段、.data 段和.bss 段)。然而，有两点主要的不同，这些不同与库模块缺乏的内容有关：

- 外部模块不包含主程序，因此没有起始地址。也就是说，在库模块中不出现标号_start:以示意连接器这是代码开始执行的地方。库模块不被设计用于自己运行，所以标号_start:在库模块中不仅是没有必要的，而且如果主程序模块中已经存在_start:，还将导致一个致命的连接器错误。
- 外部模块并不返回 Linux。如果只有主程序模块包含一个标号_start:，那么也应该只有主程序模块包含进行必要的 sys_exit INT 80h 调用的代码，以关闭程序，并将控制权交回 Linux。作为一般的经验法则，永远不要在一个过程的内部调用 sys_exit，无论它与主程序位于相同模块，还是位于外部模块库中。

首先，看一下清单 10-2，其中包含了一个名为 hexdump3 的程序。它和程序 hexdump2 完成同样的事情。从源代码的角度来看，它比 hexdump2 短了很多，因为它的大部分机制已经被甩到外部进行编码。在哪里进行外部编码呢？你现在还不知道，而且也不必知道。只要使用 EXTERN 指令列出所有地址未知的过程，NASM 就会推迟解析这些未知过程的地址。

清单 10-2：hexdump3.asm

```
; 可执行程序名    : hexdump3
; 版本            : 1.0
; 创建日期        : 4/15/2009
; 最后更新日期    : 4/20/2009
; 作者            : Jeff Duntemann
; 描述            : 一个简单的十六进制转储实用工具，
; 演示了通过 EXTERN 使用单独编译的代码库
;
; 使用这些命令来生成该程序:
;    nasm -f elf -g -F stabs hexdump3.asm
;    ld -o hexdump3 hexdump3.o <path>/textlib.o
;
SECTION .bss                ; 包含未初始化数据的段
    BUFFLEN EQU 10
    Buff resb BUFFLEN

SECTION .data               ; 包含已初始化数据的段
```

```
SECTION .text               ; 包含代码的段

EXTERN ClearLine, DumpChar, PrintLine

GLOBAL _start

_start:
        nop                 ; 这个无操作指令让 gdb 感到很高兴......
        nop
        xor esi,esi         ; 将整个字符计数器清零

; 从标准输入中读入满满一缓冲区文本:
Read:
        mov eax,3           ; 指定 sys_read 调用
        mov ebx,0           ; 指定文件描述符 0: 标准输入
        mov ecx,Buff        ; 传递要读入的缓冲区的偏移地址
        mov edx,BUFFLEN     ; 传递要在一次循环中读入的字节数
        int 80h             ; 调用 sys_read 来填充缓冲区
        mov ebp,eax         ; 保存从文件中读入的字节数以备后用
        cmp eax,0           ; 如果 eax=0, sys_read 到达了标准输入的结尾
        je Done             ; 如果相等则跳转 (与 0 相等, 通过比较)

; 为处理缓冲区这一步骤设置寄存器:
        xor ecx,ecx                ; 将缓冲区指针清零

; 搜查缓冲区, 并将二进制值转换为十六进制数字:
Scan:
        xor eax,eax               ; 将 EAX 寄存器清零
        mov al,byte[Buff+ecx]     ; 从缓冲区中取出一个字符, 放入 AL 中
        mov edx,esi               ; 复制整个计数器到 EDX 中
        and edx,0000000Fh         ; 屏蔽掉字符计数器的最低四位
        call DumpChar             ; 调用字符插入过程

; 将缓冲区指针指向下一个字符, 看看缓冲区是否已经被处理完毕:
        inc ecx                   ; 递增缓冲区指针
        inc esi                   ; 递增整个已处理字符的计数器
        cmp ecx,ebp               ; 与缓冲区中的字符个数进行比较
        jae Read                  ; 如果我们已经完成了缓冲区的处理, 继续取出
                                    更多的字符

; 看看我们是否已经处于一个长度为 16 的块的结尾, 是否需要显示一行:
        test esi,0000000Fh        ; 测试计数器中的最低 4 位是否为 0
```

```
        jnz Scan                ; 如果计数器不是 16 的余数，回去继续循环
        call PrintLine          ; ...否则，打印该行
        call ClearLine          ; 将该十六进制转储行清零
        jmp Scan                ; 继续扫描该缓冲区

; 任务全部完成！让我们结束这个晚会：
Done:
        call PrintLine          ; 打印“零头”行
        mov eax,1               ; 用于 退出系统调用的代码
        mov ebx,0               ; 返回一个零代码
        int 80H                 ; 进行内核调用
```

可以在一行中进行多个项的外部声明，中间用逗号隔开即可，比如，在 hexdump3 中：

```
EXTERN ClearLine, DumpChar, PrintLine
```

EXTERN 指令不是只能有一个。在一个模块中可能存在几条 EXTERN 指令；事实上，每个外部标识符都可能拥有自己的 EXTERN 指令。这取决于你的喜好。然而，当你有一个长长的外部标识符名单时，不要犯下如下所示的过失，那是一个错误：

```
EXTERN InitBlock, ReadBlock, ValidateBlock, WriteBlock, CleanUp,
ShowStats, PrintSummary  ; ERROR!
```

EXTERN 声明不能跨行进行。(事实上，在汇编语言中几乎没有什么能够跨越行边界，特别是使用 NASM。Pascal 和 C 程序员会经常碰到这种怪癖。) 如果你在一行中使用一个 EXTERN 进行了太多的外部声明，请在后续行中放一个额外的 EXTERN 指令。在单个模块中所能包含的 EXTERN 指令的数量是没有限制的。

为了使 hexdump3 程序连接到一个功能上可以执行的程序，我们必须为它的所有过程创建一个外部模块库。为了做到这一点，我们需要做的所有事情就是让过程和它们的数据位于正确的段中，以及进行必要的 GLOBAL 声明，如清单 10-3 所示。

清单 10-3：textlib.asm

```
;  库名            : textlib
;  版本            : 1.0
;  创建日期        : 4/10/2009
;  最后修改日期    : 4/20/2009
;  作者            : Jeff Duntemann
;  描述            : 一个可连接的面向文本过程和表的库
;  使用这些命令来生成该程序:
```

```
; nasm -f elf -g -F stabs textlib.asm
;
SECTION .bss          ; 包含未初始化数据的段
        BUFFLEN EQU 10
        Buff resb BUFFLEN

SECTION .data         ; 包含已初始化数据的段

;这里，我们采用一个可单独使用的、由两个部分组成的数据结构，
;来实现一个十六进制转储实用程序的文本行。
;第一部分显示 16 个十六进制字节，字节之间用空格隔开。
;紧接着是一个包含 16 个字符的行，二者之间用竖线字符隔开
; 因为它们是相邻的，所以可以被单独引用或者作为一个连续单位来引用。
; 请记住，如果想单独引用 DumpLin，必须在将它发送到 Linux 控制台之前，
;为其附加一个 EOL 字符，
DumpLin:    db " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "
DUMPLEN     EQU $-DumpLin
ASCLin:     db "|................|",10
ASCLEN      EQU $-ASCLin
FULLLEN     EQU $-DumpLin

; HexDigits 表用于将数字值转换为它们的十六进制等值。
; 通过一个没有缩放的半字节来进行索引：[HexDigits + eax]
HexDigits: db "0123456789ABCDEF"

; 此表允许我们为二进制数字生成文本等值。
; 通过将半字节放大 4 倍来进行索引：
; [BinDigits + ecx*4]
BinDigits: db "0000","0001","0010","0011"
           db "0100","0101","0110","0111"
           db "1000","1001","1010","1011"
           db "1100","1101","1110","1111"

; 此表用于 ASCII 字符翻译，将十六进制转储行翻译成 ASCII 部分，
; 通过 XLAT 或者普通的内存查找实现。
; 所有可打印字符直接翻译成它们本身。
; 高于 128 的字符被翻译成 ASCII 句号字符(2Eh)。
; 低 128 个字符中的不可打印字符也被翻译为 ASCII 句号字符。
DotXlat:
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
```

```
        db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
        db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
        db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
        db 60h,61h,62h,63h,64h,65h,66h,67h,68h,69h,6Ah,6Bh,6Ch,6Dh,6Eh,6Fh
        db 70h,71h,72h,73h,74h,75h,76h,77h,78h,79h,7Ah,7Bh,7Ch,7Dh,7Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh
        db 2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh,2Eh

SECTION .text                                           ; 包含代码的段
GLOBAL ClearLine, DumpChar, Newlines, PrintLine         ; 过程
GLOBAL DumpLin, HexDigits, BinDigits                    ; 数据项

;-----------------------------------------------------------------
; ClearLine: 将一个十六进制转储行清零，即设为 16 个 0 值
; 最后更新日期: 4/13/2009
; 输入参数: 无
; 返回值: 无
; 修改: 无
; 调用: DumpChar
; 描述: 十六进制转储行字符串被清为二进制 0

ClearLine:
        push edx            ; 保存主调程序的 EDX 寄存器
        mov edx,15          ; 我们将进行 16 次插入工作，从零开始计数
.Poke:  mov eax,0           ; 告诉 DumpChar 插入一个 '0'
        call DumpChar       ; 将 '0'插入该十六进制转储行字符串
        sub edx,1           ; DEC 指令不会影响 CF 标志!
        jae .Poke           ; 如果 EDX >= 0，回去继续循环
        pop edx             ; 恢复主调程序的 EDX 寄存器的值
        ret                 ; 返回

;-----------------------------------------------------------------
; DumpChar:      插入一个值到十六进制转储行字符串中
; 最后更新日期:  4/13/2009
; 输入参数:      传递即将被插入的 8 位值到 EAX 寄存器中.
;                传递该值在行中的位置(0-15)到 EDX 寄存器中
```

```
; 返回值:    无
; 修改:      EAX
; 调用:      无
; 描述:      传递到 EAX 寄存器中的值将既被放入十六进制转储部分，又被放入 ASCII
;            部分,其位置由传递到 ECX 中的值来指定,
;            如果它是一个不可打印字符，在该字符所在的位置用一个空格来表示
DumpChar:
	push ebx	; 将 EBX 寄存器的值压入堆栈，这样就不会破坏这个值了
	push edi
; 首先，我们将输入字符插入转储行的 ASCII 字符部分
	mov bl,byte [DotXlat+eax]	; 将不可打印字符翻译成 '.'
	mov byte [ASCLin+edx+1],bl	; 写入 ASCII 部分
; 接下来我们将输入字符的十六进制等值插入到该十六进制转储行的十六进制部分
	mov ebx,eax	; 将输入的字符另存一份
	lea edi,[edx*2+edx]	; 计算到字符串行中的偏移量(ECX*3)
; 查找字符的低半字节，并将其插入字符串:
	and eax,0000000Fh	; 屏蔽掉除低半字节之外的所有位
	mov al,byte [HexDigits+eax]	; 查找该半字节的字符等值
	mov byte [DumpLin+edi+2],al	; 将该字符等值写入到字符串行
; 查找字符的高半字节，并将其插入字符串:
	and ebx,000000F0h	; 屏蔽掉除第二个低半字节之外的所有位
	shr ebx,4	; 将字符的高四位移位到低四位
	mov bl,byte [HexDigits+ebx]	; 查找该半字节的字符等值
	mov byte [DumpLin+edi+1],bl	; 将该字符等值写入字符串行
; 任务完成! 让我们返回:
	pop ebx	; 恢复主调程序的 EBX 寄存器的值
	pop edi	; 恢复主调程序的 EDI 寄存器的值
	ret	; 返回主调程序

;-------------------------------------------------------------------
; Newlines:      发送 1 到 15 个换行符到 Linux 控制台
; 最后更新日期:  4/13/2009
; 输入参数:      要发送的换行符的个数，从 1 到 15
; 返回:          无
; 修改:          无
; 调用:          内核 sys_write
; 描述:          由 EDX 中的值指定数量的换行符字符(0AH)被发送到 stdout，
;                使用 INT 80h sys_write 命令。此过程演示了将常量数据放到
;                一个过程定义本身，而不是放到.data 段或者.bss 段。

Newlines:
	pushad	;保存主调程序的所有寄存器的值
```

```
        cmp edx,15          ; 确保主调程序的请求没有超过 15 个
        ja .exit            ; 如果超过,什么都不做就退出
        mov ecx,EOLs        ; 将 EOLs 表的地址放入 ECX 寄存器中
        mov eax,4           ; 指定 sys_write
        mov ebx,1           ; 指定 stdout
        int 80h             ; 进行内核调用
.exit   popad               ; 恢复主调程序的所有寄存器的值
        ret                 ; 返回
EOLs: db 10,10,10,10,10,10,10,10,10,10,10,10,10,10,10

;-------------------------------------------------------------------
; PrintLine:   通过 INT 80h sys_write 来显示十六进制转储行
; 最后更新日期: 4/13/2009
; 输入参数:    无
; 返回:        无
; 修改:        无
; 调用:        内核 sys_write
; 描述:        通过使用 INT 80h sys_write 指令，将十六进制转储行字符串
;              显示到标准输出。
PrintLine:
        pushad                  ; 将所有的通用寄存器压入堆栈
        mov eax,4               ; 指定 sys_write 调用
        mov ebx,1               ; 指定文件描述符 1: 标准输出
        mov ecx,DumpLin         ; 传递字符串行的偏移量
        mov edx,FULLLEN         ; 传递字符串行的大小
        int 80h                 ; 进行内核调用以显示该字符串行
        popad                   ; 弹出所有的通用寄存器
        ret                     ; 返回!
```

有两行全局标识符声明，每行都有自己的 GLOBAL 指令。按照我自己的工作习惯，我将过程和已命名数据项分开声明，让它们每个占用一行：

```
GLOBAL ClearLine, DumpChar, Newlines, PrintLine    ; 过程
GLOBAL DumpLin, HexDigits, BinDigits               ; 数据项
```

任何即将被导出 (换句话说，即将被做成在模块之外可用) 的过程或数据项都必须在 GLOBAL 指令后面的行内进行声明。不一定过程中的所有内容都要声明为 global。事实上，一种处理复杂性以及避免特定类型的漏洞的方法是努力思考，严格限制其他模块能够在它们的同伴模块中“看到”的内容。模块拥有只能从该模块内部引用的、私有的过程和命名数据项。事实上，私有类型是这些项的缺省设置：只要不将它们声明为 global 即可。

注意，所有声明为 global 的项必须在它们被定义之前就声明为 global。实际上，这意味着在对这些过程进行任何真正定义之前，我们都需要在.text 段的顶部将其声明为全局过程。类似地，在对所有全局类型的已命名数据项进行定义之前，我们必须在.data 段中对它们进行声明。

我们也可以从模块中导出等值，虽然这是一项 NASM 汇编编译器的创新，并不一定对于所有的汇编编译器都正确。只需将一个与等值相关联的标号放到一个 EXTERN 定义列表中，然后其他模块就能看到和使用这个等值了，虽然等值必须被当作像 Pascal 常数一样的只读值使用。

10.4.3 连接库文件到程序中

对于本书在前面提供的所有示例程序而言，生成文件都非常简单。例如，这里是 hexdump2 程序的生成文件：

```
hexdump2: hexdump2.o
   ld -o hexdump2 hexdump2.o
hexdump2.o: hexdump2.asm
   nasm -f elf -g -F stabs hexdump2.asm
```

连接器调用将 hexdump2.o 转换为可执行文件 hexdump2，这就是它必须做的所有事情。添加库文件会使情况变得稍微有点复杂。现在连接器必须要做一些实际的多文件连接工作。在连接器调用中，主程序的可连接文件名的后面添加了一些额外的.o 格式的库文件。在一个连接步骤中可以有任意(合理)数量的.o 文件。为了生成 hexdump3，我们只需要两个.o 文件。这里是 hexdump3 的生成文件：

```
hexdump3: hexdump3.o
    ld -o hexdump3 hexdump3.o ../textlib/textlib.o
hexdump3.o: hexdump3.asm
    nasm -f elf -g -F stabs hexdump3.asm
```

在连接器调用行中,textlib.o 文件被简单地放在程序本身的.o 文件后面。在生成文件的前半部分，有一个“缩水”：库文件采用的是包含 hexdump3 项目的路径的相对目录。将“../textlib/”放在文件名 textlib.o 的前面允许连接器“穿过”Linux 文件系统，进入项目目录去寻找库文件本身。否则，你必须将 textlib.o 文件放到 hexdump3 的同一目录下，或者将其复制到 usr / lib 目录下的某个目录中，其中 usr / lib 是默认的搜索路径。

一旦生成完毕，建议将其存放到位于 usr/ lib 下的某个目录中。当你仍在从

事库文件开发时，最好把它存放在一个跟你的其他项目目录位于同一棵目录树下的项目目录中，这样你就可以修复一些现在还没有发生，直到你已经使用它一段时间来生成其他的程序之后才可能遇到的漏洞，以及增加一些功能了。

10.4.4 太多过程和太多库的危险

过犹不及。我曾经见到过包含数百个文件的代码库，其中每个文件都包含一个单个的过程。而且这些也不是独立的过程。它们在一个厚厚的执行网络中彼此互相调用，这是在源代码级非常难以追踪的，特别是如果你从其他人那里继承了这样一个库，你必须掌握(通常很快)库文件的实际工作的实现机制。如果没有非常详细的文本文档，就没有一个“从某一高度的视图”来帮助你掌握什么过程从哪些地方调用什么过程。如果库文件来自其他地方，并且可以像“黑盒子”一样使用，那可能不是一个灾难，但是我仍然想知道我所使用的每一个库文件是如何工作的。

以这种方式创建库，有一个正当理由：当你将一个库文件连接到某个程序时，整个库都被添加到可执行文件中，即使这些过程和数据定义永远不会从主程序中引用。如果每一个过程都被分开来单独汇编成自己的小.o 文件，那么连接器将只把那些将在程序中实际调用的(因此被程序执行的)过程添加到程序中。

这在很大程度上取决于代码追求的最终目标。如果目标是最小的可执行文件，这是重要的。在汇编语言世界中有一些大陆(尤其是那些与嵌入式系统相关的)，其中的每一个字节都具有重要的意义，那些从不运行的死代码将会为支撑它运行的低端硬件增加不必要的开销。

在一台普通的、拥有 3GB 内存和 250GB 硬盘的 Linux 电脑上，汇编语言代码的大小将不会带来任何问题。如果这是你的代码即将运行的地方，那么拥有更少的库文件以及更多的易于理解的源代码会更好些，即使最终的可执行文件的代码达到了几千字节，也永远不会与 CPU 进行面对面的交锋。

10.5 自定义过程的艺术

过程创建不仅仅是单纯地将一段代码从某个程序中“切片”出来，然后制造一个包含 CALL 指令和 RET 指令的“三明治”，除此之外还包含更多的内容。过程的主要目的是通过将那些服务于公共目标的指令集合在一起，将其做成一个命名实体，因而使代码更加容易维护。不要忘了火星人，以及他们如何在 1977

年绑架我那个倒霉的 APL 文本格式化实用程序。可维护性在软件设计中可能是块最难啃的骨头，而且可维护性完全取决于可理解性。整个自定义过程库的理念就是使你的代码——主要是对你而言，但是也很有可能对任何其他可能继承或者尝试使用你的代码的人而言——易于理解。

因此，在这一节，我们将要谈论如何构思一个过程以及如何创造一个过程，在此期间一定要将代码的可维护性牢记心间。

10.5.1 可维护性和可重用性

过程的一个最重要目的是：通过用一个描述性的名称来代替一个机器指令序列来管理程序的复杂性。其次是代码重用。每次开始一个新的项目都从头开始编写一些通用机制是没有意义的。只需编写实现 em 算法的过程一次，并将该过程编写得很棒，然后我们就永远使用这个 em 过程。

这两个目的是相辅相成的。代码重用主要通过以下几种方式帮助我们实现代码的可维护性：

- 从维护所有项目的广度角度来看，重用意味着较少的代码总量。
- 重用维护了你花费在调试方面的时间和精力。
- 重用迫使你一直在项目中保持一定的编码约定(因为你的库需要它)，这就给了你的项目成员一种“家族相似性”，从而在离开一段时间后，你仍然能够比较容易地掌握它们。
- 重用意味着你将拥有较少的、只是以稍微不同的方式完成相同事情的代码序列。

最后一点非常微妙，但是却非常重要。当你在调试程序时，经常需要参考程序中每个部分的工作原理。你希望这种认识对于你编写的每一个程序都是独一无二的，但是事实却并非如此。记忆是不确切的，除了非常类似的事情之外，单独的记忆往往会模糊在一起。(快速回答：那是 1972 年的黑斑羚还是 1973 年的黑斑羚？)在编程中，细节问题非常关键，而且在汇编语言编程中，恰恰存在很多的细节。如果你为三个只有很小的不同之处的程序从头开始编写三次 RefreshText 过程，那么，当盯着某个 RefreshText 过程时，你可能会依赖于对另一个 RefreshText 过程的实现的理解。而这些相似但却不相同的过程的创建时间越久远，你就越有可能混淆它们，进而不得不浪费时间整理每个过程操作它们的小怪癖。

但是，如果只有一个 RefreshText 过程，那么就只有一个对 RefreshText 的理

解需要回想。上面提到的所有有关重用的优点可以归结为一句话，即：通过简单地减少必须管理的复杂性的数量来管理复杂性。

10.5.2 确定哪些代码应该成为一个过程

那么，什么时候一个指令块应该被拉出来，并且做成一个过程呢？不存在硬性且快速的规则，但是也有一些是值得探讨的有益启发：

- 寻找经常在一个程序内部发生的行为。
- 寻找可能不会在任何单个程序中发生很多次，但却往往以相同的方式在许多或者大多数程序中发生的行为。
- 当程序变得越来越大(这里的“大”，我的意思是超出教程类演示程序；比方说 1000 行左右)时，查找可被做成过程的功能模块，以便使主程序的整个执行流程变得更加容易理解。(稍后会有更多与此相关的讲解。)
- 在程序中寻找那些为了响应不可控的力量(如数据规格、第三方库，等等)而可能随时间变化的行为，并将这些行为隔离到过程中。

总之：想得大一点，想的长远一点。你才不会永远是个初学者。尽量预先考虑你在将来某个时刻的编程劳动，创建一些具有一般用途的过程。这里的“一般”是指不仅对于你现在刚好工作的单个程序有用，而且对你未来编写的程序有用。

对于过程而言，如果被经常调用，没有最小尺寸的限制。非常简单的过程——甚至是那些只包含 4 条或者 5 条指令的过程——都不会把自己隐藏在很大的复杂性之中。它们为那些经常使用的行为赋予特定的描述性名称，这本身是有价值的。它们还可以为创建体积更大和功能更强的过程提供标准的基础构建块。尽管如此，如果一个简短的代码序列(5～10 条指令)只在一个中等规模的只包含数百机器指令的程序中调用一次也许两次，那么它将是一个贫穷的过程“候选人”，除非它是一个在今后的过程中可能被重用的人选。那样的话，它可以属于一个代码库，而且代码只能存在于某个过程之中，不能单独位于库中。

对于过程而言，没有最大尺寸，而且在某些情况下，创建一些较大的过程很有意义，只要它们能够服务于既定目标。请记住，过程并不总是需要在库中。当程序变得足够大时，为了增强可理解性，需要将其分解为功能模块，你可能会发现定义那种只调用一次的大型过程是很有用的。一个包含 1000 行代码的汇编语言程序可能被分割成七八个相当大的过程序列。每个过程被设计为只打算在主程序里调用一次，但是它们可以让主程序变得更短，更容易掌握，并且更加明确地

指明主程序正在做什么：

```
Start: call Initialize          ; 打开指定文件，创建缓冲区
       call OpenFile            ; 打开目标数据文件
Input: call GetRec              ; 从打开的文件中取出一条记录
       cmp eax,0                ; 在文件读取过程中测试 EOF
       je Done                  ; 如果我们已经碰到了 EOF，就到了关闭程序的时间
       call ProcessRec          ; 处理该条记录
       call VerifyRec           ; 验证该条记录修改后的数据
       call WriteRec            ; 将修改后的记录写出到文件中
       jmp Input                ; 回去，再次重复做这件事情
Done:  call CloseFile           ; 关闭打开的文件
       call CleanUp             ; 删除临时文件
       mov eax,1                ; 指定退出系统调用
       mov ebx,[StatusCode]     ; 将状态码传递给操作系统
       int 80h                  ; 返回 Linux
```

当汇编语言程序开始接近一千行时，这个干净的、具有良好的可读性的程序提供了一个必要的“处于某一高度的视图”。请记住，火星人总是躲在附近的某个地方，渴望将你的程序变成不可读的“象形”文字。

对抗火星人的其他武器甚至连过程威力的一半都达不到。

10.5.3 使用注释标头

随着时间的推移，你会发现，为了管理程序的复杂性，自己已经创造了几十个甚至数百个过程。大多数高级语言供应商为他们的编译器提供的“罐装”库过程在 NASM 中并不存在。一般来说，当需要某些功能时，你必须自己编写它。

当你已经独自编写了很多例程时，维护这样的例程列表并不是一项容易的任务。必须将每个单独过程相关的本质事实形成文档，否则你将会很快忘记它们，或者将它们记错并按照错误的信息采取行动。(由此产生的漏洞往往很难找到，因为你确认已经记住了所有关于该过程需要记住的内容。毕竟，这些过程是你自己编写的！)

我强烈建议为每一个自己编写的过程增加一个注释标头，无论这个过程的功能多么简单。这样的标头应至少包含以下信息：

- 过程的名称。
- 最后修改日期。
- 每个入口点的名称，如果该过程拥有多个入口点的话。

- 过程所做的事情。
- 主调程序必须向其传递哪些数据它才能正常工作。
- 哪些数据(如果有的话)被该过程返回，以及这些数据被返回到哪个地方(例如，到寄存器 ECX 中)。
- 过程修改了哪些寄存器或者数据项。
- 哪些其他过程被该过程调用(如果有的话)。
- 在编写使用该过程的程序代码时需要牢记的所有注意事项。

除此之外，注释标头中的其他信息有时对我们也是有帮助的：

- 过程的版本，如果使用版本控制的话。
- 过程的创建日期。
- 编写该过程的人的姓名(如果你正在处理共享于一个团队内的代码的话)。

一个典型的可行的过程标头看起来像这样：

```
;-----------------------------------------------------------------
; LoadBuff: 通过 INT 80h sys_read 从标准输入中读入数据以填充缓冲区
; 最后更新日期: 4/15/2009
; 输入参数: 无
; 返回值: 将读入的字节数存入寄存器 EBP 中
; 修改内容: ECX, EBP, Buff
; 调用: 内核 sys_read
; 描述:从标准输入中读入满满一缓冲区数据 (BUFFLEN 个字节)
; 使用 INT 80h sys_read 将其放入 Buff.缓冲区
; 清零偏移计数器 ECX,因为我们正开始往一个新的缓冲区装数据。主调程序必须测试
EBP 中的值:
; 如果 EBP 包含零则返回,说明我们遇到了标准输入的 EOF.
; 如果 EBP 中的返回值比零小，则表明发生了某种类型的错误。
```

注释标头并不代表你不用在过程内部的各行添加注释！正如我曾经多次重申的那样，在每一行包含机器指令助记符的代码右侧添加一个简短的注释，和(在较长过程中)为过程添加一个描述该过程的每一个主要的功能的注释块，是一个不错的主意。

10.6　Linux 控制台下的简单光标控制

作为一个从汇编过程到汇编宏之间的“序曲”，我想花点时间讲解从程序内控制 Linux 控制台显示的细节。让我们回到为乔的餐馆显示的小小的廉价便利小

吃店广告。为它增加一点刺激性：首先清空 Linux 控制台，然后将该广告文字显示在清空后的显示器的中央。我将把同样的程序展示两次，第一次将几个部分表示为过程，第二次将同样的部分表示为宏。

首先是包含过程的程序，如清单 10-4 所示。

清单 10-4：eatterm.asm

```
; 可执行文件名   : eatterm
; 版本           : 1.0
; 创建日期       : 4/21/2009
; 最后更新日期   : 4/23/2009
; 作者           : Jeff Duntemann
; 描述           : 一个运行在 Linux 下的、简单的汇编程序，使用 NASM 2.05，
;                  展示了使用转义序列实现简单的“全屏”文本输出
;
; 使用以下命令生成：
;    nasm -f elf -g -F stabs eatterm.asm
;    ld -o eatterm eatterm.o
;
section .data                     ; 包含已初始化的数据
SCRWIDTH: equ 80                  ; 默认情况下我们假设 80 个字符宽
PosTerm: db 27,"[01;01H"          ; <ESC>[<Y>;<X>H
POSLEN: equ $-PosTerm             ; 术语位置字符串的长度
ClearTerm: db 27,"[2J"            ; <ESC>[2J
CLEARLEN equ $-ClearTerm          ; 术语清除字符串的长度
AdMsg: db "Eat At Joe's!"         ; 广告消息
ADLEN: equ $-AdMsg                ; 广告消息的长度
Prompt: db "Press Enter: "        ; 用户提示信息
PROMPTLEN: equ $-Prompt           ; 用户提示信息的长度

; 此表为我们提供了 0 到 80 之间的数字的以两位表示的 ASCII 位元(digit)。
; 我们不是通过计算来产生插入终端控制字符串中的 ASCII 数字，
; 而是在表中查找它们，一次取出两个 ASCII 位元到一个 16 位寄存器中，例如 DX，
; 然后将该寄存器的值放到终端控制字符串 PosTerm 的适当位置。参见 GotoXY。
;  如果打算在更大的、超过 80*80 的控制台上工作，
; 必须在 Dights 的结尾添加额外的 ASCII 位元编码。
; 注意，此处显示的代码只能工作在最大 99*99 的控制台上。
Digits: db "0001020304050607080910111213141516171819"
        db "2021222324252627282930313233343536373839"
        db "4041424344454647484950515253545556575859"
        db "606162636465666768697071727374757677787980"
```

```
SECTION .bss                    ; 包含未初始化的数据

SECTION .text                   ; 包含代码的段

;-------------------------------------------------------------------------
; ClrScr:       清空 Linux 控制台
; 最后更新日期:  4/21/2009
; 输入参数:      无
; 返回值:        无
; 修改:          无
; 调用:          内核 sys_write
; 描述:          发送预定义的控制字符串 <ESC>[2J 到控制台,它将清空整个显示器
ClrScr:
        push eax                        ; 保存有关寄存器
        push ebx
        push ecx
        push edx
        mov ecx,ClearTerm               ; 传递终端控制字符串的偏移地址
        mov edx,CLEARLEN                ; 传递终端控制字符串的长度
        call WriteStr                   ; 发送终端控制字符串到控制台
        pop edx                         ; 恢复有关寄存器
        pop ecx
        pop ebx
        pop eax
        ret                             ; 返回

;-------------------------------------------------------------------------
; GotoXY:       将 Linux 控制台光标定位到(X,Y) 坐标处
; 最后更新日期:  4/21/2009
; 输入参数:      X 存放到 AH, Y 存放到 AL
; 返回值:        没有
; 修改:          PosTerm 终端控制序列字符串
; 调用:          内核 sys_write
; 描述:          为传递到 AL 和 AH 的 X,Y 坐标准备一个终端控制字符串
;               并且调用 sys_write 将控制台光标定位到(X,Y)位置。
;               在调用 GotoXY 之后，写文本到控制台将从(X,Y)位置开始显示文本。
GotoXY:
        Pushad                  ; 保存主调程序的寄存器
        xor ebx,ebx             ; 清零 EBX
        xor ecx,ecx             ; ECX 寄存器操作同上
; 插入 Y 的数字:
        mov bl,al               ; 将 Y 的值放入缩放术语 EBX 寄存器中
```

```
        mov cx,word [Digits+ebx*2]      ; 取出十进制数字放到 CX 寄存器中
        mov word [PosTerm+2],cx         ; 将数字插入控制字符串中
; 插入 X 的数字：
        mov bl,ah                       ; 将 X 的值放入缩放术语 EBX 寄存器中
        mov cx,word [Digits+ebx*2]      ; 取出十进制数字放到 CX 寄存器中
        mov word [PosTerm+5],cx         ; 将数字插入到控制字符串中
; 发送控制序列到标准输出：
        mov ecx,PosTerm                 ; 传递控制字符串的地址
        mov edx,POSLEN                  ; 传递控制字符串的长度
        call WriteStr                   ; 发送控制字符串到控制台
; 扫尾，返回：
        Popad                           ; 恢复主调程序的寄存器
        ret                             ; 返回

;-----------------------------------------------------------------
; WriteCtr: 发送一个字符串到一个 80 个字符宽的 Linux 控制台的中央位置
; 最后更新日期: 4/21/2009
; 输入参数: 传递 Y 值到 AL 中，字符串地址到 ECX 中，字符串长度到 EDX 中
; 返回值: 无
; 修改: PosTerm 终端控制序列字符串
; 调用: GotoXY, WriteStr
; 描述: 在一个 80 列宽的 Linux 控制台显示区域中央显示一个字符串
;     为传入的字符串的长度计算 X,
; 然后调用 GotoXY 和 WriteStr 将该字符串发送到控制台

WriteCtr:
        push ebx                  ; 保存主调程序的 EBX
        xor ebx,ebx               ; 将 EBX 清零
        mov bl,SCRWIDTH           ; 将显示器宽度值加载到 BL 中
        sub bl,dl                 ; 取出显示器宽度和字符串长度的差值
        shr bl,1                  ; 将该差值除以 2 以求取 X 的值
        mov ah,bl                 ; GotoXY 需要位于 AH 中的 X 的值
        call GotoXY               ; 为显示指定信息定位光标
        call WriteStr             ; 将该字符串写到控制台
        pop ebx                   ; 恢复主调程序的 EBX
        ret                       ; 返回

;-----------------------------------------------------------------
; WriteStr: 发送一个字符串到 Linux 控制台
; 最后更新日期: 4/21/2009
; 输入参数: 传递字符串地址到 ECX 中，字符串长度到 EDX 中
; 返回值:  无
```

```
; 修改: 无
; 调用: 内核 sys_write
; 描述: 通过一个 sys_write 内核调用显示一个字符串到 Linux 控制台，
;

WriteStr:
        push eax                ; 保存相关的寄存器
        push ebx
        mov eax,4               ; 指定 sys_write 调用
        mov ebx,1               ; 指定文件描述符 1: 标准输出
        int 80H                 ; 进行内核调用
        pop ebx                 ; 恢复相关的寄存器
        pop eax
        ret                     ; 返回

        global _start           ; 连接器需要这个来找到入口点!

_start:
        nop                     ; 这个无操作指令让 gdb 感到非常高兴

; 首先我们清空终端显示器
        call ClrScr

; 然后我们传递广告消息到这个 80 个字符宽的控制台的中央:
        mov al,12               ; 纵坐标为第 12 行
        mov ecx,AdMsg           ; 传递消息的地址
        mov edx,ADLEN           ; 传递消息的长度
        call WriteCtr           ; 将它显示到控制台上

; 为“按 Enter 键……”这一提示信息定位光标:
        mov ax,0117h            ; 将 X,Y = 1,23 作为一个单个的十六进制值放入 AX
        call GotoXY             ; 定位光标

;显示提示信息“按 Enter 键……”:
        mov ecx,Prompt          ; 传递提示信息的偏移地址
        mov edx,PROMPTLEN       ; 传递提示信息的长度
        call WriteStr           ; 将提示信息发送到控制台

; 等待用户敲击 Enter 键:
        mov eax,3               ; 调用 sys_read 的代码
        mov ebx,0               ; 指定文件描述符 0:标准输入
        int 80H                 ; 进行内核调用
```

```
; ...我们任务完成
        Exit: mov eax,1          ; 用于退出系统调用的代码
        mov ebx,0                ; 将零作为返回码
        int 80H                  ; 进行内核调用
```

这里也有一些新的机制。到目前为止，我在本书中提供的所有程序只是简单地顺序发送文本行到标准输出，然后控制台依次显示它们，一行一行地不断滚动。

这可以是非常有用的，但是它不是我们能够达到的最佳效果。回到第 6 章，我简要地介绍了一种可以控制 Linux 控制台的方式：在从程序到标准输出"旅行"的文本流中嵌入一些转义字符序列。如果你不记得这段内容了，回去重读一下会很有用，因为我不会在这里对它进行深刻地回顾。

一个用于控制控制台的转义序列的最简单的例子是将整个控制台的显示清空(从根本上说，空格字符)。在 eatterm 程序中，这个序列是一个字符串变量，叫作 ClearTerm：

```
ClearTerm: db 27,"[2J" ; <ESC>[2J
```

一个转义序列的长度为 4 个字符。它始于 Esc 键，一个不可打印字符，我们通常通过它的十进制 ASCII 值 27 来描述它。(或十六进制值，即 1Bh。)紧跟在 ESC 字符后面的是 3 个可打印字符[2J。它们是可打印字符，但是却不能被打印出来，因为它们跟在 Esc 键后面。控制台留意 ESC 字符，并且根据一项非常大且复杂的方案，以特殊的方式解释跟在 ESC 字符后面的任何字符。特殊的序列代表对控制台的特殊命令，比如这个序列将清除显示内容。

在转义序列结尾，并没有结束标记用于指示该序列已完成。控制台知道对于某个字母的每一个转义序列，其中包括每个序列有多长，而且不会发生歧义。在示例 ClearTerm 序列中，控制台知道，当它看到字符 J 时，这个序列完成。然后，它清除显示，恢复为继续把程序发给它的字符(原样)显示到标准输出。

在发送转义序列到控制台方面，你不需做什么特别的事情。转义序列通过一个 INT 80h 调用被发送到标准输出，就像所有其他文字一样。通过在程序的.text 段认真安排 DB 指令，你可以在可打印文本中间嵌入转义序列。请记住，即使转义序列不在控制台上显示，当你通过 INT 80h 传递一个文本序列的长度给 sys_write 时，它们仍然被计入文本的长度。

用于清除显示的转义序列很容易理解，因为它总是相同的，而且总是做完全相同的事情。定位光标位置的转义序列则有些麻烦，因为它需要参数来指定光标

即将移到的(X，Y)位置。这些参数中的每一个都是一个 ASCII 形式的两位数字文本值，它们必须在该序列发送到标准输出之前，被程序嵌入到文本序列中。在 Linux 控制台上移动光标的所有技巧均都涉及将参数 X 和 Y 嵌入转义序列中。

在 eatterm 中定义的默认序列被叫作 PosTerm：

```
PosTerm: db 27,"[01;01H" ; <ESC>[<Y>;<X>H
```

如同 ClearTerm 一样，它始于一个 ESC 字符。字符[和字符 H 之间是两个参数。Y 值在前面，用一个分号与 X 值分开。请注意，这些都不是二进制数字，而是两个 ASCII 字符代表的数值型数字——在本例中，为 ASCII 码 48(0)和 ASCII 49(1)。你不能只是插入二进制值" 1 "到转义序列。控制台并不将二进制值 1 理解为 ASCII 49。代表 X 和 Y 位置的二进制值必须首先转换为它们的 ASCII 等值，然后再插入到转义序列中。

这就是过程 GotoXY 所做的事情。通过在表中查找对应的 ASCII 字符来将二进制值转换为它们的 ASCII 等值。Digits 表提供了 0～80 之间的数值的 2 位 ASCII 表示形式。10 以下的数值第一位为 0，如 01、02、03，等等。下述代码指出了 GotoXY 的转换原理：

```
; 插入 Y 的 数字:
    mov bl,al                      ; 将 Y 的值放入 EBX 中用作缩放比例
    mov cx,word [Digits+ebx*2]     ; 取出十进制数字到 CX 中
    mov word [PosTerm+2],cx        ; 将数字插入控制字符串中
; 插入 X 的 数字:
    mov bl,ah                      ; 将 X 的值放入 EBX 中用作缩放比例
    mov cx,word [Digits+ebx*2]     ; 取出十进制数字到 CX 中
    mov word [PosTerm+5],cx        ; 将数字插入到控制字符串中
```

X、Y 的值被传入 AL 和 AH 两个 8 位寄存器中。每一个值都被放入一个已清零的 EBX 寄存器中，并作为从 Digits 处开始的有效地址术语。因为 Digits 表的每个元素的大小是 2，所以我们必须将偏移地址放大两倍。

诀窍(如果有的话)是既将这两个 ASCII 数字放入一个内存引用，又将其放入 16 位寄存器 CX 中。对于 CX 中的两个 ASCII 数字，我们又同时将它们插入到转义序列的适当位置。Y 的值开始于字符串内偏移量为 2 的地方，而 X 的值开始于偏移量为 5 的地方。

一旦 PosTerm 字符串已经为某一特定的 X，Y 坐标对修改完毕，该字符串就被发送到标准输出，并且被控制台作为控制光标位置的转义序列来解释。下一个发送到控制台的字符将出现在这个新的光标位置，后续字符将在随后的位置出

现，直至另一个光标控制序列被发送到控制台。

当运行发出光标控制代码的程序时，请确保控制台窗口比光标将会采用的最大的 X 值和最大的 Y 值大一些；否则，行将会折叠起来，在预期显示的地方什么都不会显示。eatterm 程序中有一个 `Digits` 表，正好达到 80 * 80。如果想在一个更大的显示范围内工作，你必须将 `Digits` 表所包含的两位数字的 ASCII 等值扩展到 99。鉴于该表的创建和引用方式，你只能取出 2 位数值，因此对于此处显示的代码而言，你被限定在一个 99 字符 * 99 字符的控制台中。

控制台控制注意事项

这一切都听起来很棒——但是它并不像听起来那么好。最基本的控制序列，像清除显示器和移动光标，可能是非常普遍的，并且可能在任何你能找到的 Linux 控制台上同等地工作。当然，它们能够工作在为基于 Debian 的 Linux 发行版而设计的两种最流行的控制台终端实用程序——GNOME Terminal 和 Konsole 上。

不幸的是，Unix 终端和终端控制的历史，是一个带有斑点的故事；更加先进的控制台控制功能可能不支持那些序列，或者从一个控制台实现到另一个控制台实现有所不同。为了确保一切正常，你的程序将不得不检测控制台，以找出它所支持的终端规范，然后再发出相应的转义序列。

这是一件令人羞愧的事情。在 Konsole 中，下面的转义序列将把控制台的背景转换成绿色：

```
GreenBack: db 27,"[42m"
```

这在 konsole 中是正确的。至于这个序列的通用性如何，以及其他控制台是否喜欢这个序列，我真的不知道。众多其他的控制台控制命令也是如此，通过这些命令，你可以打开和关闭个人电脑键盘的发光二极管、改变前景色、显示下划线，等等。更多关于这方面(以简洁 Unix 风格)的资料可在 Linux 手册中根据关键字“console_codes”找到，我鼓励你去尝试，同时铭记不同的控制台(特别是在非 Linux 的 Unix 实现上)可能对不同的转义序列有不同的反应方式。

不过，控制控制台的输出并不是最糟糕的。控制台编程的最高境界是创造全屏文本应用程序，它能够在控制台上“绘制”一个表格，包括数据入口字段，并允许用户通过使用 TAB 键从一个字段域跳到另一个字段域，而且可在每个字段域中输入数据。这在 Linux 中极为困难，因为需要通过一些名为原始模式的方式访问控制台键盘上的单个按键。仅仅说明原始模式的工作原理就将花费一章的大

部分篇幅，而且会涉及很多我在本书中没有空间讲解的比较先进的 Linux 话题。

处理控制台的标准 UNIX 方式是一个名为 ncurses 的 C 库，并且虽然 ncurses 可以从汇编中进行调用，但是它实在是一个既臃肿又丑陋的家伙。汇编程序员的更好选择是一个比较新的、专门为 NASM 汇编语言而编写的库，名为 LinuxAsmTools。它作者是杰夫·欧文斯，它几乎能做所有 ncurses 能做的事情，而且没有 C 的硬性约定和令人讨厌的东西。LinuxAsmTools 是免费的，你可以在 http://linuxasmtools.net/上找到它。

10.7　创建和使用宏

有不止一种方法可以将一个汇编语言程序分割成更易于管理的块儿。过程是最明显的方式，当然也是最容易理解的方式。从过程中调用和返回的机制被直接内置在 CPU 中，而且独立于任何一个的特定汇编产品。

当今大多汇编编译器提供了另外一种复杂性管理工具：宏。宏是一个完全不同的种类。但是，过程是通过使用直接建立在指令集中的 CALL 指令和 RET 指令来实施的，而宏则是一种汇编编译器的技巧，它不依赖于任何特定的指令或指令组。

简单地说，宏是一个代表某些文本行序列的标号。这个文本行序列可以(但不一定)是一个指令序列。当汇编编译器在源代码文件中遇到宏标号时，它用该宏标号所代表的文本行来取代该宏标号。这叫作宏扩展，因为宏的名称(只占一个文本行)被替换为几行文本，然后这些文本再被编译，就好像它们一直出现在源代码文件中一样。(当然，宏不一定要有多行文本。它可能只有一行文本——但是这样做之后，由使用它所带来的优点将会少很多！)

宏与高级语言(如 Pascal)中的包含(include)文件有一些相似之处。在 Borland Pascal 和较新的 Pascal，如 FreePascal 中，一个包含命令可能看上去像这样：

```
{$I ENGINE.DEF}
```

当遇到上述包含(include)命令时，编译器到磁盘上查找名为 ENGINE.DEF 的文件。然后打开该文件，并将其包含的文本输入到源代码文件中，放在包含命令所在的位置。然后编译器处理那些行，就像它们一直在源代码文件那里一样。

你可以把宏看作是一个内置到源代码文件中的包含文件。它是一系列定义一次的文本行，拥有一个描述名称，然后我们就可以根据需要，仅仅通过它的名字即可将其重复“扔进”源代码中。

这个过程如图 10-4 所示。存储在磁盘上的源代码有一个宏定义，它放在%MACRO 和%ENDMACRO 之间的括号中。在文件的后面部分，这个宏的名字出现了好几次。当汇编编译器处理这个文件时，它将这个宏定义复制到某个内存缓冲区中。当编译从磁盘中读入的文本时，汇编编译器将包含在宏中的文本放到每一处出现该宏名字的地方。磁盘上的文件并不会受到影响；宏扩展只发生在内存中。

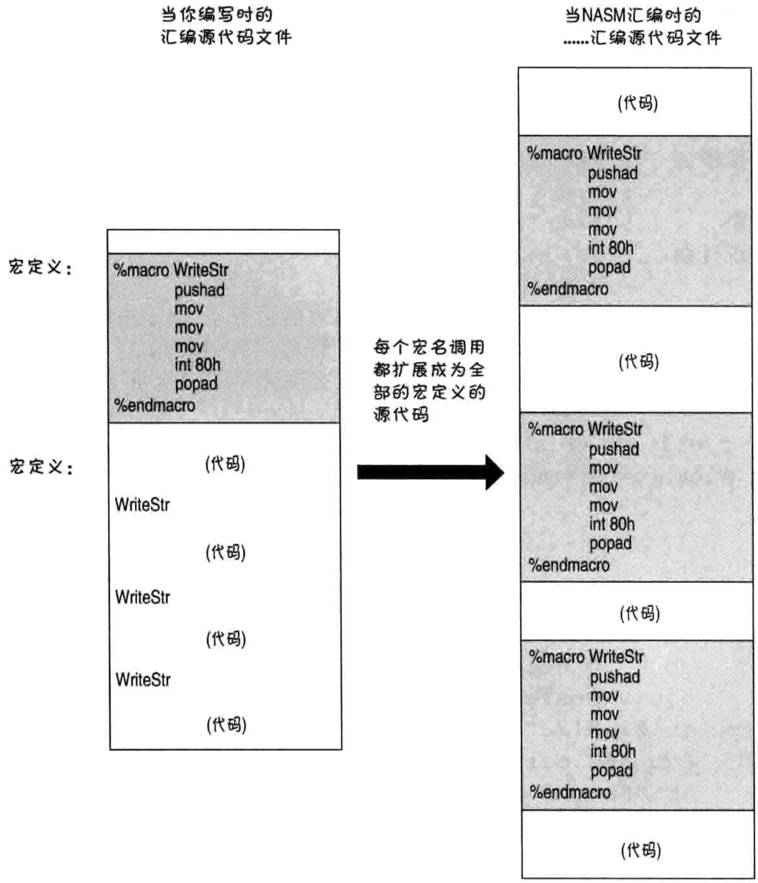


图 10-4　宏工作原理

10.7.1　宏定义机制

宏定义看起来有点像过程定义，框在 NASM 的一对特殊指令：%MACRO 和%ENDMACRO 之间。请注意%ENDMACRO 指令是在宏的最后一行后面的行中。

不要误将%ENDMACRO 看成是一个标志着宏的最后一行的标号。

与过程相比，宏的一个小缺点是它只有一个入口点。毕竟，宏是被插入到程序执行流中的一个代码行序列。你不能调用宏，而且也不能从它返回。CPU 像运行普通的指令序列一样来运行它。

许多或者说大部分程序可以被小心地表示为一个宏。在清单 10-5 中，我所采用的程序来自清单 10-4，并且将所有的过程转换成了宏，以便你可以看出这两者之间的差异。

清单 10-5：eatmacro.asm

```
; 可执行程序名    : eatmacro
; 版本            : 1.0
; 创建日期        : 4/21/2009
; 最后更新日期    : 4/23/2009
; 作者            : Jeff Duntemann
; 描述            :一个简单的用汇编语言编写的运行于 Linux 下的程序,
;                  使用 NASM 2.05,演示了转移序列的使用,
;                  通过使用宏而没有过程来实现简单的“全屏”文本输出
;                  通过以下命令生成:
; nasm -f elf -g -F stabs eatmacro.asm
; ld -o eatmacro eatmacro.o
;
;
section .data                              ; 包含已初始化数据的段

SCRWIDTH:  equ 80                          ; 默认情况下，我们假设 80 个字符宽
PosTerm:   db 27,"[01;01H"                 ; <ESC>[<Y>;<X>H
POSLEN:    equ $-PosTerm                   ; 术语位置字符串的长度
ClearTerm: db 27,"[2J"                     ; <ESC>[2J; 清除显示
CLEARLEN   equ $-ClearTerm                 ; 术语清除字符串的长度
AdMsg:     db "Eat At Joe's!"              ; 广告消息
ADLEN:     equ $-AdMsg                     ; 广告消息的长度
Prompt:    db "Press Enter: "              ; 用户提示信息
PROMPTLEN: equ $-Prompt                    ; 用户提示信息的长度

; 这张表为我们提供了 0 到 80 之间的数字的两位 ASCII 位元(ASCII digit).
; 我们没有通过计算来生成插入终端控制字符串的 ASCII 数值,
; 而是从表中查找它们，然后一次取出两个 ASCII 位元到一个 16 位寄存器中，如 DX,
; 然后再将其插入终端控制字符串 PosTerm 的合适位置。参见 GotoXY.
; 如果打算在一个比 80*80 大的控制台上工作,
; 必须在 Digits 表的结尾添加额外的 ASCII 位元编码
```

```
; 请记住，这里显示的代码只适合于最大 99*99 的控制台上。
Digits: db "0001020304050607080910111213141516171819"
        db "2021222324252627282930313233343536373839"
        db "4041424344454647484950515253545556575859"
        db "6061626364656667686970717273747576777879 80"

SECTION .bss                ; 包含未初始化数据的段

SECTION .text               ; 包含代码的段

;-----------------------------------------------------------------
; ExitProg:     终止程序并返回 Linux
; 最后更新日期: 4/23/2009
; 输入参数:     无
; 返回值:       无
; 修改内容:     无
; 调用:         内核 sys_exit
; 描述:         调用 sys_edit 来终止程序并将控制返回 Linux
%macro ExitProg 0
       mov eax,1            ; 退出系统调用的代码
       mov ebx,0            ; 返回一个零代码
       int 80H              ; 进行内核调用
%endmacro

;-----------------------------------------------------------------
; WaitEnter: 等待用户在控制台按 Enter 键
; 最后更新日期: 4/23/2009
; 输入参数: 无
; 返回值: 无
; 修改内容: 无
; 调用: 内核 sys_read
; 描述: 调用 sys_read 来等待用户在控制台上输入一个换行符

%macro WaitEnter 0
       mov eax,3            ; sys_read 的代码
       mov ebx,0            ; 指定文件描述符 0: Stdin
       int 80H              ; 进行内核调用
%endmacro

;-----------------------------------------------------------------
; WriteStr:     发送字符串到 Linux 控制台
; 最后更新日期: 4/21/2009
```

```
; 输入参数:     在 %1 中给出字符串地址, 在 %2 中给出字符串长度
; 返回值:       无
; 修改内容:     无
; 调用:         内核 sys_write
; 描述:         通过一个 sys_write 内核调用来显示一个字符串到 Linux 控制台

%macro WriteStr 2           ; %1 = 字符串地址; %2 = 字符串长度
       push eax             ; 保存相关寄存器
       push ebx
       mov ecx,%1           ; 将字符串地址放到 ECX 中
       mov edx,%2           ; 将字符串的长度到 EDX 中
       mov eax,4            ; 指定 sys_write 调用
       mov ebx,1            ; 指定文件描述符 1: Stdout
       int 80H              ; 进行内核调用
       pop ebx              ; 恢复相关寄存器
       pop eax
       %endmacro

;-----------------------------------------------------------------
; ClrScr:       清空 Linux 控制台
; 最后更新日期: 4/23/2009
; 输入参数:     无
; 返回值:       无
; 修改内容:     无
; 调用:         内核 sys_write
; 描述:         发送预定义的控制字符串 <ESC>[2J 到控制台,
;               该控制字符串将清空整个显示区域
%macro ClrScr 0
       push eax             ; 保存相关寄存器
       push ebx
       push ecx
       push edx
; 使用 WriteStr 宏将控制字符串写入控制台:
       WriteStr ClearTerm,CLEARLEN
       pop edx              ; 恢复相关寄存器
       pop ecx
       pop ebx
       pop eax
%endmacro

;-----------------------------------------------------------------
; GotoXY:       定位 Linux 控制台光标到一个 (X,Y) 位置
```

```
; 最后更新日期:  4/23/2009
; 输入参数:      X 放入%1, Y 放入 %2
; 返回值:        无
; 修改内容:      PosTerm 终端控制序列字符串
; 调用:          内核 sys_write
; 描述:          为传递到 AL 和 AH 中的 X,Y 坐标准备一个终端控制字符串
;                并调用 sys_write 将控制台光标定位到那个(X,Y)位置
;                如果在调用 GotoXY 之后向控制台写文本，该文本将会在位置(X,Y)处显示

%macro GotoXY 2                 ; %1 是 X 的值; %2 是 Y 的值
       pushad                   ; 保存主调程序的寄存器
       xor edx,edx              ; 将寄存器 EDX 清零
       xor ecx,ecx              ; ECX 寄存器操作同上
;插入 Y 的数字
       mov dl,%2                ; 将 Y 的值放入偏移地址术语 EDX 寄存器中
       mov cx,word [Digits+edx*2]   ; 取出十进制数字放到 CX 寄存器中
       mov word [PosTerm+2],cx      ; 将数字插入到控制字符串中
; 插入 X 的数字:
       mov dl,%1                ; 将 X 的值放入用于计算偏移地址的 EDX 寄存器中
       mov cx,word [Digits+edx*2]   ; 取出十进制数字放到 CX 寄存器中
       mov word [PosTerm+5],cx      ;将数字插入到控制字符串中
; 发送控制序列到标准输出:
       WriteStr PosTerm,POSLEN
;处理善后工作，返回:
       popad                        ; 恢复主调程序的寄存器
%endmacro

;--------------------------------------------------------------------
; WriteCtr: 发送一个字符串到一个 80 个字符宽的 Linux 控制台的中央
; 最后更新日期: 4/23/2009
; 输入参数: Y 的值在 %1 中, 字符串的地址在%2 中, 字符串的长度在%3 中
; 返回值: 无
; 修改内容: PosTerm 终端控制序列字符串
; 调用: GotoXY, WriteStr
; 描述: 显示一个字符串到 Linux 控制台，位于一个 80 列宽的显示区域的中央;
;       为传入的字符串的长度计算 X, 然后调用 GotoXY 和 WriteStr 将该字符串
;       发送到控制台

%macro WriteCtr 3       ; %1 =行; %2 = 字符串地址; %3 =字符串长度
       push ebx                 ; 保存主调程序的 EBX 寄存器
       push edx                 ; 保存主调程序的 EDX 寄存器
       mov edx,%3               ; 加载字符串的长度到 EDX 寄存器中
```

```
        xor ebx,ebx              ; 清零 EBX 寄存器
        mov bl,SCRWIDTH          ; 加载屏幕宽度值到 BL 中
        sub bl,dl                ; 计算显示器宽度和字符串长度的差值
        shr bl,1                 ; 将该差值除以 2 以求取 X 的值
        GotoXY bl,%1             ; 为显示定位光标
        WriteStr %2,%3           ; 将字符串写入控制台
        pop edx                  ; 恢复主调程序的 EDX 寄存器
        pop ebx                  ; 恢复主调程序的 EBX 寄存器
%endmacro

global _start                    ; 连接器需要这个来找到入口点！

_start:
        nop                      ; 这个无操作指令让 gdb 感到十分高兴

;首先我们清除终端显示
        ClrScr
;然后我们发送广告消息，使其在 80 个字符宽的控制台上居中显示：
        WriteCtr 12,AdMsg,ADLEN
;为显示"Press Enter"提示信息定位光标：
        GotoXY 1,23
; 显示 "Press Enter" 提示信息：
        WriteStr Prompt,PROMPTLEN
; 等待用户按 Enter 键：
        WaitEnter
; 大功告成！
        ExitProg
```

比较一下 eatmacro 中的宏和 eatterm 中的与它们等值的过程。它们已经摆脱了 RET 指令(和对于那些调用其他的宏的宏而言，它们的 CALL 指令)，但是在很大程度上，它们几乎由完全相同的代码组成。

宏被简单地通过提及它们的名字来进行调用。再次重申一下，不要使用 CALL 指令。只要将宏的名字放在一行中即可：

```
ClrScr
```

汇编编译器会处理其余的事情。

10.7.2 定义带参数的宏

宏在很大程度上是一种直接的文字替换技巧，但是文字替换也有一些有趣

的，有时甚至非常有用的窍门。其中之一是当宏被调用时，具有能够将参数传递给宏的能力。

例如，在 eatmacro 中有一个对于具有 3 个参数的 WriteCtr 宏的调用：

```
WriteCtr 12,AdMsg,ADDLEN
```

字面常量 12 被传递给宏，用于指定要在屏幕中心显示的文本所在的屏幕行。在本例中，是从顶部数第 12 行。你可以用 3 或 16 或任何低于目前在 Linux 控制台中可显示的行的数目的其他数字来取代 12。(如果你尝试将光标定位到不出现在控制台内的行中，结果将很难预测。通常情况下，文本从屏幕的最底行开始出现。)其他两个参数传递要显示的字符串的地址和长度。

再次重申，宏的参数是汇编编译器的高级神器。它们不被压入堆栈或设置到一个共享内存区域中，(像常见的那样)或任何类似的地点。参数只是一些为那些通过宏参数传递给宏的实际值(叫作实参)而预留的占位符。

让我们仔细观察一下 WriteCtrmacro，看看它是如何工作的：

```
%macro WriteCtr 3       ; %1 = 行号; %2 =字符串地址; %3 = 字符串长度
push ebx                ; 保存主调程序的 EBX 寄存器
push edx                ; 保存主调程序的 EDX 寄存器
mov edx,%3              ; 加载字符串的长度到 EDX 寄存器
xor ebx,ebx             ; 清零 EBX 寄存器
mov bl,SCRWIDTH         ; 加载屏幕的宽度值到 BL 中
sub bl,dl               ; 计算显示器宽度和字符串长度的差值
shr bl,1                ; 将该差值除以二以求取 X 的值
GotoXY bl,%1            ; 为显示指定信息定位光标
WriteStr %2,%3          ; 将该字符串写到控制台
pop edx                 ; 恢复主调程序的 EDX 寄存器
pop ebx                 ; 恢复主调程序的 EBX 寄存器
%endmacro
```

那么，参数从何处来？这是 NASM 从根本上与微软的 MASM 不同的另一个领域。MASM 允许你使用符号名称——如“行”或“字符串长度”之类的单词——来代表参数。NASM 依赖于一个简单的系统，该系统在宏定义中就声明了参数的个数，然后通过宏内部的参数编号而不是符号名称来引用每个参数。

在宏 WriteCtr 的定义中，宏名称后面的 3 表明汇编编译器将要寻找三个参数。这个数字必须存在——即便是宏根本不带任何参数，该值为 0。每个宏都必须有参数数量。在宏的定义中，参数被通过参数的编号来进行引用。因此，当名为“WriteCtr”的宏被引用时，“%1”表明第一个参数被使用；“%2”表明第二

个参数被使用，从左往右计数；“%3”表明第三个参数被使用，等等。

传递到形参(parameter)的实际值被称为实参(argument)。不要混淆形参和它的实际值。如果你了解 Pascal，它们之间的差异完全类似于形式参数和实际参数之间的差异。

宏的形参对应于 Pascal 的形式参数，而宏的实参对应于 Pascal 的实际参数。宏的形参是在宏定义的行中紧跟在宏名称后面的标号。宏的实参是在调用宏的行中所指定的值。

宏的形参是一种类型的标号，它们可以在宏内部的任何地方进行引用——但是只能在宏的内部引用。在 WriteCtr 这个宏中，形参%3 被作为一条 MOV 指令的操作数来引用。因此，传递给宏的形参%3 的实参将被加载到寄存器 EDX 中。

宏的实参可以被作为形参传递给其他的宏。这也是当 WriteCtr 宏调用 WriteStr 宏时，发生在 WriteCtr 宏内部的情况。WriteStr 宏需要两个形参，并且 WriteCtr 将自身的形参%2 和形参%3 作为 WriteStr 的实参传递到 WriteStr 宏中。

10.7.3 宏调用机制

我们可以将一个字面常量的值作为一个实参传递给宏，就像在 eatmacro 程序中，将 row 这个值传递到 WriteCtr 宏中一样。我们还可以传递一个寄存器的名称作为实参。以下是一个对 WriteCtr 宏的完全合法且合理的调用方法：

```
mov al,4
WriteCtr al,AdMsg,ADLEN
```

在 WriteCtr 中，NASM 用 AL 寄存器的名字取代了%1 形参，所以

```
GotoXY bl,%1          ; 定位光标用于显示指定内容
```

变成了

```
GotoXY bl,al
```

千万要注意，所有管理指令操作数的一般规则在宏中都适用。形参%1 只能容纳一个 8 位的实参，因为最终%1 被加载到一个 GotoXY 里面的 8 位寄存器中。你不能合法地传递寄存器 EBP 或 CX 到 WriteCtr 宏的形参%1 中，因为不能直接移动一个 32 位或 16 位寄存器到一个 8 位寄存器中。

类似的，你可以将一个带括号的地址作为实参来传递：

```
WriteCtr [RowValue],AdMsg,ADLEN
```

当然，这里假设 RowValue 是一个被定义为 8 位数据项的已命名变量。如果在一条需要使用 32 位实参的指令中使用宏形参(如同 WriteCtr 宏的形参%3 和%2)，那么你也可以传递代表 32 位地址的标号或 32 位的数字值。

当宏被调用时，它的参数由逗号隔开。NASM 按照从左到右的顺序，依次将实参放到宏的形参中。如果只传递两个实参到一个带有三个形参的宏中，你可能会从汇编编译器中得到一条错误信息，这取决于你如何引用那个未被填充的形参。如果你传递较多的参数到一个具有较少的形参的宏中，那些额外的实参将被忽略。

10.7.4　宏内部的局部标号

eatmacro.asm 中所包含的宏设计得相当简单且显而易见。它们根本没有包含任何转移指令，但是宏中的代码可以使用条件转移指令和无条件转移指令，就像程序体和过程中能做的一样。然而，对于宏内部使用的标号，存在一个重要的问题：在汇编语言程序中的标号必须是唯一的，然而，从本质上来讲，一个宏被调用了多少次，它的源代码就被重复了多少次。这意味着将有标号重复的错误信息……除非宏的标号都被视为局部标号。

局部项对于定义它们的框架之外的代码是没有任何意义的。宏的局部标号在宏的定义之外是不可见的，这意味着，它们将不能被引用，除非是从%MACRO 到%ENDMACRO 范围之内的代码引用。

在宏定义内部的所有标号都被认为是宏的局部标号，并且被汇编编译器进行了特殊的处理。下面是一个例子；它是我在前面提供的一块代码的宏转换，用于将缓冲区中的小写字符强制转换为大写字符：

```
%macro UpCase 2              ; %1 = 缓冲区地址; %2 =缓冲区中的字符
mov edx,%1                   ; 将缓冲区的偏移地址放入 edx 寄存器中
mov ecx,%2                   ; 将缓冲区中的字节个数放入 ecx 寄存器中
%%IsLC: cmp byte [edx+ecx-1],'a'     ; 是否比'a'小?
jb %%Bump                    ; 如果不是小写字符，跳过
cmp byte [edx+ecx-1],'z'        ; 是否大于'z'?
ja %%Bump                       ; 不是小写字符，跳过
sub byte [edx+ecx-1],20h        ; 强制将缓冲区中的字节转换为大写字符
%%Bump: dec ecx                 ; 递减字符计数器的值
jnz %%IsLC                      ; 如果缓冲区中还有字符，继续重复处理过程
```

```
%endmacro
```

宏内部的标号通过以两个百分号(%%)开头来作为局部标号使用。当在宏中标识一个位置时，这个局部标号后面应该跟着一个冒号。当用作一个 jump 指令或 call 指令(如 JA,JB,以及前面的 JNZ)的操作数时，局部标号的后面不跟冒号。重要的是要明白，除非标号 IsLC 和 Bump 都被通过添加前缀%% 而作为宏的局部标号，否则，将会在程序中出现多个标号实例，(假设宏被不止一次调用)，汇编编译器将在第二个和以后每个调用中产生一个标号重复的错误。

由于实际上在程序中标号必须是独一无二的，所以 NASM 采用宏局部标号，例如%%Bump，并由它生成一个程序中独一无二的标号。它通过采用前缀"..@"来将一个 4 位数字和标号的名称相连。每次宏被调用时，NASM 都将改变这个数，从而为每一个宏内部的局部标号产生一个独一无二的同义词。例如，%%Bump 标号，在一次给定的调用中，可能会成为.. @ 1771.Bump，而且这个数将在每次该宏被调用时发生变化。这种情况发生在幕后，你几乎意识不到它的发生，除非阅读了由 NASM 生成的代码转储清单文件。

10.7.5 像包含文件一样的宏库文件

正如过程可以被收集起来，做成程序的外部模块库一样，宏也可以被收集起来，建立一个宏库。宏库文件就是一个包含了宏源代码的文本文件。不像被收集到模块中的过程，宏库文件被单独编译，并且每次当程序被编译时必须通过汇编编译器。这是宏的一个一般性问题，不仅仅针对收集到库中的宏。通过将代码分解成宏来管理其复杂性的程序，其编译速度将比那些已经被分割成单独的已编译模块的程序更慢一些。这在今天看来已经是一个比 20 年前小得多的问题了，但是对于非常大的项目而言，它可以影响其生成速度。

我们可以通过将宏库包含到程序源代码文件中来使用它。使用%INCLUDE 指令能够实现这一点。%INCLUDE 指令出现在宏库的名称之前：

```
%include "mylib.mac"
```

从技术上来讲，这条语句可能出现在源代码文件中的任何地方，但是请记住，所有的宏在它们被调用之前，必须已经完全定义。出于这个原因，在任何一个可能的宏库文件调用发生之前，在源代码文件的.text 段的顶部附近使用%INCLUDE 指令，这是一个好主意。

如果包含到某个程序中的宏文件没有与该程序位于相同的目录，需要提供一

个完整的路径名作为%INCLUDE 指令的一部分：

```
%include "../macrolibs/mylib.mac"
```

否则，NASM 可能无法找到宏文件，并且提交一个错误信息。

10.7.6 宏 vs. 过程：优点和缺点

宏有一些超过过程的优点。其中之一就是速度。执行和控制进入一个过程的 CALL 指令和从一个过程中退出的 RET 指令均需要时间。而在一个宏中，这两条指令一个也不使用。只有执行宏的实际工作的指令被执行，因此宏的工作可以尽快地完成。

然而，这个较快的速度是需要付出代价的，那就是它需要使用额外的内存，尤其是当宏被调用了许多次时。注意，在图 10-4 中，三次对 WriteStr 宏的调用在内存中一共生成了 18 条指令。如果宏被创建为一个过程，那么在过程体中将只需 6 条指令，外加一条 RET 指令，并且我们只需使用 3 条 CALL 指令就可以完成同样的工作。也就是说，全部功能如果采用过程实现，将一共只需要 10 条指令，而如果采用宏实现，则需要 18 条指令。如果宏被调用 5 次或 7 次以上，差异会更大。每当一个宏被调用时，其所有的指令都将在程序中被再复制一次。

在较短的程序中，这可能不会带来什么问题，但是在某些情况下，代码必须尽可能快速地执行——比如，在图形驱动程序中——宏由于能够消除过程调用和返回带来的开销而经常倍受到青睐。这里是一个容易理解的简单折中方案：如果为了速度，考虑宏；如果为了紧密性，考虑过程。

另一方面，除非你在真正编写某些绝对性能依赖的程序，例如图形驱动程序，否则这种权衡已经小到了无足轻重的程度。对于普通的软件而言，一个面向过程的实施和一个面向宏的实施的大小差异可能只有两三千个字节，速度的差异可能察觉不到。在现代的 CPU 上，任何给定软件的性能都难以预料，海量存储设备和存储系统使得程序的大小变得远不如对上一代那么重要了。如果你正在试图决定在某一给定情况下是采用过程还是采用宏，请主要考虑规模或速度以外的其他因素。

举例来说，我总觉得宏密集型软件调试起来要更加困难。软件工具不一定处理得好宏。gdb 调试器的 Insight 组件并不在它的源代码窗口显示宏扩展文本。Insight 不是被专门设计用来纯粹考虑汇编调试(像大多数 Unix 工具一样，Gdb 非常偏好 C)的，当你步入一个宏时，源代码高亮显示会停止，一直到执行从宏中再

次出现。因此你不能像单步调试过程或程序代码那样步入宏的代码。Gdb 将仍然一如既往地从控制台窗口进行调试，但是，与从 Insight 中的可视化操作相比，控制台调试是一个非常痛苦的过程。

最后，还有另一个难以解释的与宏相关的问题，它也是我对它们感觉很不舒服的重要原因：如果使用的宏太多，代码看起来将不再像是汇编语言。让我们再看看 eatmacro.asm 程序的主程序的一部分：

```
ClrScr
WriteCtr 12,AdMsg,ADLEN
GotoXY 1,23
WriteStr Prompt,PROMPTLEN
WaitEnter
ExitProg
```

这就是全部的主程序。整个事情已经被宏调用归纳了。这是汇编语言，抑或是——天哪！——一个 BASIC 的方言呢？

我承认，我在这里用宏调用取代了整个主程序，目的就是为了证明这一点，但是创造如此多的宏，以至于你的汇编程序开始像某些奇怪的高级语言，这种情况肯定是有可能的。

困难的真相是，宏能够澄清一个程序正在做什么，或者，如果使用过多，它们完全可以掩盖事物在本质层面上的工作原理。我在自己的项目中，使用宏仅仅为了减少非常重复的指令序列所带来的混乱，尤其是诸如内核调用之前，设置寄存器之类的指令。毕竟整个汇编编程的意义是要培养一种对于在软件遇到 CPU 的地方发生了什么事情的全面了解。凡是阻碍这一目标的东西，都应谨慎地、巧妙地、(最重要的是)有节制地使用，否则你倒不如学习 C 了。

第 11 章

字符串奏鸣曲

神奇的字符串指令

到目前为止，本书已经介绍了汇编语言工作的大部分重要方面，涵盖了几乎所有种类的机器指令。但是，仍然有一类指令没有涉及，在我看来，它们很可能是所有指令中最有趣的：x86 字符串指令。

在 x86 指令集的所有指令中，只有它们有权在内存里一次处理长序列的字节、字或者双字。(在汇编语言中，内存中的任意连续字节序列可以被认为是一个字符串——而不仅仅是人类可读的字符序列。) 更加神奇的是，一些字符串指令有权以一种非常紧凑的方式处理大量的字节序列：像执行单条指令一样执行一个完整的指令循环，而且完全在 CPU 之内完成。

在这一章中，我们将主要讲解一下汇编语言的字符串，并且涵盖更多与编程和 Linux 调试相关的主题。

11.1 汇编语言字符串的概念

单词有时令我们失望，因为它们不能像磁铁捡起铁屑那样一如既往地反映它们所代表的含义。字符串这个词就是这方面的一个“典范”。这意味着它在几乎所有的计算机编程中都意味着大致相同的事情，但是针对某一主题也有很多细小的变化。如果了解 Pascal 字符串(像我一样)，你会发现，当使用 C/C++，Python、Basic，或者(尤其是)汇编进行编程时，你所知道的内容并不完全适用。

所以这里有一个采用尽可能简单的术语的描述：字符串是在内存中的任意一组连续的字节，可以为操作系统允许的任意大小。(对于 Linux 而言，它可以很大。)汇编语言中的字符串的主要概念界定是：它的成员字节刚好排成一排，没有中断。

这很重要。大多数高级语言从几个方面建立字符串的概念。起源于 Turbo Pascal 的 Pascal 将字符串视为一个单独的数据类型，并在字符串的开始处有一个长度计数器以表明该字符串包含多少个字节。在 C 中，字符串在它的前面并没有长度字节。相反，当遇到一个值为 0 的二进制字节时，我们认为一个 C 字符串结束。这一点在汇编工作中非常重要，其中大部分与 C 以及 C 字符串处理机制所在的标准 C 函数库密切相关。在 Basic 中，字符串被存储在名为*字符串空间*的地方，该空间拥有很多与字符串相关的内置代码机制，用以管理字符串空间并处理字符串数据的纵深操作。

当你开始使用汇编语言工作时，必须要放弃所有高级语言的东西。汇编字符串只是一块连续的内存区域。它们从某一指定的地址开始，持续一些数量的字节，然后停止。没有长度计数器以显示字符串包含了多少个字节，也没有标准边界字符(例如二进制的 0)以表明一个字符串从哪里开始或者结束。你当然可以编写能够分配和操纵 Turbo Pascal 风格的字符串或 C 风格的字符串的汇编语言例程，但是为了避免混淆，你必须把这些例程所操作的数据当作 Pascal 或者 C 字符串，而不是汇编语言字符串。

11.1.1　彻底颠覆你的“字符串感觉”

汇编字符串没有边界值或者长度指示器。它们可以包含任意值，包括二进制 0。事实上，你真的不能再根据内存中的特定区域范围来思考字符串了。相反地，你应该根据定义字符串的寄存器的值来思考它们。

这与在诸如 Pascal 之类的语言中思考字符串相比，有点从内往外思考的感觉，但是它的确起作用：当你将一个寄存器设置为指向一个字符串时，你就拥有了一个字符串。而一旦你指向了一个字符串，该字符串的长度将由存放在寄存器 ECX 中的值来指定。

这是关键，并且冒着涉嫌重复的危险，我再重申一遍：汇编字符串是完全由存放在寄存器中的值来进行定义的。在 CPU 的芯片上，硬编码了一组关于字符串和寄存器的假设。当执行一条字符串指令(我将在稍后进行描述)时，CPU 将使用这些假设来确定它读取或者写入哪些内存区域。

11.1.2　源字符串和目标字符串

在 x86 汇编工作中，有两种类型的字符串。源字符串是你从中读取数据的字

符串。目标字符串是你写入数据的字符串。两者之间的区别只是一个寄存器的问题。源字符串和目标字符串也可以重叠。事实上，同一内存区域在同一时间里可以既是一个源字符串，又是一个目标字符串。

这里是在 32 位保护模式下执行字符串指令时，CPU 对字符串所做的假设：

- ESI 寄存器指向源字符串。
- EDI 寄存器指向目标字符串。
- 这两种字符串的长度都等于存放在 ECX 寄存器中的值。至于这个长度如何对 CPU 起作用，取决于具体的指令及该指令的使用方法。
- 来自一个源字符串或者去往一个目标字符串的数据必须要么从寄存器 EAX 开始旅行，要么在寄存器 EAX 中结束旅行，要么通过寄存器 EAX。

CPU 能够同时识别源字符串和目标字符串，因为 ESI 和 EDI 可以存放两个相互独立的值。但是，由于只有一个 ECX 寄存器，所以，当同时使用源字符串和目标字符串时，它们的长度必须相等，比如：在复制源字符串到目标字符串之类的操作中。

一种记住源字符串和目标字符串之间差异的办法是通过它们的偏移地址寄存器。ESI 的意思是“已扩展的源索引”，而 EDI 的意思是“已扩展的目标索引”，当然，“已扩展的”的意思是：与它们祖先也就是 x86 计算时期的 16 位寄存器 SI 和 DI 相比，它们的大小是 32 位。

11.1.3　虚拟文本显示屏

巩固所有关于字符串背景资料的最佳办法，就是观察一些正在工作中的字符串指令。在程序清单 11-1 中，我使用字符串指令实现了一个有趣的机制：一个 Linux 控制台下的简单的虚拟文本显示屏。

回到在 PC 上采用 DOS 环境进行实模式编程的日子，那时我们可以畅通无阻地进入显卡上的视频显示刷新缓冲区。如果我们在构成显示缓冲区的内存区域内编写一个 ASCII 字符或者字符串，砰！相关的文字符号瞬间就出现在了屏幕上。在这本书的早期版本中，我利用那种直接访问显示设备的机制，提出了一套演示 x86 字符串指令的有用的显示例程。

但是，在 Linux 下，这已不再成为可能。图形显示缓冲区仍然在那里，但是它现在是 Linux 内核的财产，用户空间应用程序不能对它进行写入操作，甚至连从它直接读取数据都不可以。

使用汇编语言为 Linux 控制台编写文本模式的应用程序已经根本不可能再像 DOS 下那么容易了。在第 10 章中，我已经解释了(非常)简单的控制台终端控制可以通过使用 INT 80 指令向控制台写入转义序列来实现。然而，除了两三条最简单的命令外，终端实现的变化使得使用“赤裸裸的”转义序列有点冒险。一个给定序列可能对某一终端意味着一件事情，而对于另一个终端则意味着完全不同的另外一件事情。像 ncurses 一样的终端控制库竭尽全力检测和适应多种类型的外部终端。但是，能做那件事情的代码不是你一个下午就能够草草编成的，事实上它是一个太大的主题，以至于不适合在像本书这样的书籍中进行详细介绍。

不过，我们可以拉出其中的一些技巧，并且通过拉出它们来学会一些东西。其中之一就是在内存中创建我们自己的文字视频刷新缓冲区，并将其作为一个已命名变量，定期通过 INT 80h 指令写出到 Linux 控制台上。自 DOS 时代以来，电脑的运行速度已经变得非常迅速，但是文字视频缓冲区却不大。一个 25 × 80 的文本显示缓冲区只有 2000 个字符长，我们只需一个简单的 INT 80h sys_write 调用，就可以将整个内容发送到控制台上。缓冲区能够以至少人类视觉感知不到的速度瞬间出现。

将文本放到缓冲区中是一种简单问题，无非就是计算缓冲区中给定行列位置的地址，然后再将 ASCII 字符值写入到从该地址开始的缓冲区变量中。在每一次修改缓冲区变量后，我们都能够通过使用 INT 80h 指令将整个缓冲区写到控制台上来更新控制台显示。疲倦不堪的专家们可能称之为“蛮力”(并且它的功能远不及 ncurses 通用强大)，但是它很容易理解。虽然它不允许控制字符颜色或者属性(下划线、闪烁等)，但是却能使我们对于 x86 字符串指令有一个很好的基本了解。

观察一下清单 11-1 中的代码。在下面的几节中，我们将一块一块地浏览它。

清单 11-1：vidbuff1.asm

```
; 可执行名字      : VIDBUFF1
; 版本            : 1.0
; 创建日期        : 5/11/2009
; 最后修改日期    : 5/14/2009
; 作者            : Jeff Duntemann
; 描述            : 一个使用汇编语言为 Linux 编写的程序，
;                   使用的是 NASM 2.05,
;                   通过“伪造”全屏内存映射文本 I/O
;                   演示了字符串指令操作
```

```
; 可通过以下命令生成：
; nasm -f elf -g -F stabs vidbuff1.asm
; ld -o vidbuff1 vidbuff1.o

SECTION .data              ;包含已初始化数据的段。
   EOL equ 10              ; Linux 的行结束字符。
   FILLCHR equ 32          ; ASCII 空格字符
   HBARCHR equ 196         ;如果这个字符不能显示，使用破折号字符。
   STRTROW equ 2           ;图形开始的行

; 数据集只是一张由长度为 1 个字节的数字构成的表：
   Dataset db 9,71,17,52,55,18,29,36,18,68,77,63,58,44,0

   Message db "Data current as of 5/13/2009"
   MSGLEN equ $-Message

; 此转义序列将清除控制台终端的显示，
; 并将文本光标放到几乎所有的 Linux 控制台的原点(1,1)处：
   ClrHome db 27,"[2J",27,"[01;01H"
   CLRLEN equ $-ClrHome           ;用于清除屏幕内容的字符串的长度

SECTION .bss                      ; 包含未初始化数据的段

   COLS equ 81                    ; 行的长度+1 个用于表示行结束的 EOL 字符
   ROWS equ 25                    ; 显示的行数
   VidBuff resb COLS*ROWS         ; 将缓冲区的大小调整为适合存放 ROWS * COLS
                                    个字符

SECTION .text                     ; 包含代码的段

global _start                     ; 连接器需要这个来找到入口点！

; 此宏清除 Linux 控制台终端的显示内容，并使用一个预定义的转义序列，
; 将光标位置设置到(1,1)。
%macro ClearTerminal 0
   Pushad                         ; 保存所有的寄存器
   mov eax,4                      ; 指定 sys_write 调用
   mov ebx,1                      ; 指定文件描述符 1：标准输出
   mov ecx,ClrHome                ; 传递错误信息的偏移地址
   mov edx,CLRLEN                 ; 传递消息的长度
   int 80H                        ; 进行内核调用
   popad                          ; 恢复所有的寄存器
```

```
%endmacro

;-------------------------------------------------------------------
; Show: 显示一个文本缓冲区到 Linux 控制台
; 最后更新日期: 5/13/2009
; 输入参数: 无
; 返回值 : 无
; 修改内容:无
; 调用: Linux sys_write
; 描述: 通过 sys_write 发送缓冲区 VidBuff 到 Linux 控制台。
;        发送到控制台的字节数通过计算行等值乘以列等值得到。

Show: pushad                    ; 保存所有的寄存器
    mov eax,4                   ; 指定 sys_write 调用
    mov ebx,1                   ; 指定文件描述符 1: 标准输出
    mov ecx,VidBuff             ; 传递缓冲区的偏移地址
    mov edx,COLS*ROWS           ; 传递缓冲区的长度
    int 80H                     ; 进行内核调用
    popad                       ; 恢复所有的寄存器
    ret                         ; 返回!

;-------------------------------------------------------------------
; ClrVid:       将所有文本缓冲区清为空格字符，并替换所有的行结束字符 EOL
; 最后更新日期: 5/13/2009
; 输入参数:     无
; 返回值 :      无
; 修改:         VidBuff, DF
; 调用 :        无
; 描述:         用预定义字符填充缓冲区 VidBuff (FILLCHR)
;               并在每行结尾放一个 EOL 字符,
;               缓冲区 VidBuff 中每 COLS 个字节就有一个行结束符
ClrVid: push eax             ; 保存所有主调程序的寄存器
    push ecx
    push edi
    cld                      ; 清除 DF;采用上内存(up-memory)方式计数
    mov al,FILLCHR           ; 将缓冲区过滤器字符放入 AL
    mov edi,VidBuff          ; 指向缓冲区中的目标索引
    mov ecx,COLS*ROWS        ; 将存储的字符的个数放到 ECX 寄存器中
    rep stosb                ; 在缓冲区中"爆炸”字符
;   缓冲区被清除，现在我们需要在每一行重新插入 EOL 字符:
    mov edi,VidBuff          ; 再次指向缓冲区中的目标字符串
    dec edi                  ; 从 VidBuff 缓冲区中的字符 0 开始 EOL 位置计数
```

```
    mov ecx,ROWS            ; 将行数放入计数器寄存器
PtEOL: add edi,COLS         ; 将列数加到 EDI 寄存器上
    mov byte [edi],EOL      ; 存储 EOL 字符到行的结尾
    loop PtEOL              ; 如果还有更多行，回去继续循环
    pop edi                 ; 恢复所有主调程序的寄存器
    pop ecx
    pop eax
    ret                     ; 返回!

;-----------------------------------------------------------------
; WrtLn: 将一个字符串写到一个文本缓冲区的基于 1 的 X，Y 位置
; 最后更新日期: 5/13/2009
; 输入参数: 字符串的地址被传递到 ESI 寄存器中
;           基于 1 的 X 位置 (行 #)被传递到 EBX 寄存器中
;           基于 1 的 Y 位置 (列 #)被传递到 EAX 寄存器中
;           以字符为单位的字符串的长度被传递到 ECX 寄存器中
; 返回值 :无
; 修改内容: VidBuff, EDI, DF
; 调用:无
; 描述: 使用 REP MOVSB 指令从 ESI 中指定的地址
;       复制一个字符串到文本缓冲区 VidBuff 中的 X,Y 位置
WrtLn: push eax         ; 保存我们修改的寄存器
    push ebx
    push ecx
    push edi
    cld                 ; 清除 DF 以便以上内存的方式执行写入操作
    mov edi,VidBuff     ; 与缓冲区地址一起，加载目标索引
    dec eax             ; 调整 Y 的值，使之向下减 1，用于地址计算
    dec ebx             ; 调整 X 的值，使之向下减 1，用于地址计算
    mov ah,COLS         ; 将显示器的宽度移到 AH 中
    mul ah              ; 执行 8 位乘法: AL*AH ，将结果存入 AX 中
    add edi,eax         ; 将 Y 在 vidbuff 缓冲区中的偏移地址加到 EDI 寄存器上
    add edi,ebx         ; 将 X 在 vidbuff 缓冲区中的偏移地址加到 EDI 寄存器上
    rep movsb           ; 在缓冲区中“爆炸”字符
    pop edi             ; 恢复我们修改的寄存器
    pop ecx
    pop ebx
    pop eax
    ret                 ; 返回!

;-----------------------------------------------------------------
; WrtHB: 在文本缓冲区的(X,Y)位置处产生一个水平线条
```

```
; 最后更新日期: 5/13/2009
; 输入参数: 基于 1 的 X 位置 (行 #)被传递到 EBX 寄存器中
;           基于 1 的 Y 位置 (列 #)被传递到 EAX 寄存器中
;           以字符为单位的水平条的长度被传递到 ECX 寄存器中
; 返回值: 无
; 修改: VidBuff, DF
; 调用: 无
; 描述:  在视频缓冲区 VidBuff 的从 1 开始的、由传递到 EBX 和 EAX 中的 X,  Y 的值
; 指定的位置写入一个水平线。
;       该水平线由 HBARCHR 等值字符构成。缺省情况是字符 196;
;       如果你的终端不能显示那个值(你需要使用 IBM 850 字符集),
;       将 HBARCHR 中的值修改为 ASCII 破折号或者某些你的终端支持的其他字符.

WrtHB: push eax         ; 保存我们修改的寄存器
    push ebx
    push ecx
    push edi
    cld                 ; 清除 DF 以便以上内存方式写数据
    mov edi,VidBuff     ; 将缓冲区的地址放入目标寄存器
    dec eax             ; 将 Y 的值向下调整 1, 用于地址计算
    dec ebx             ; 将 X 的值向下调整 1, 用于地址计算

    mov ah,COLS         ; 将屏幕宽度放入 AH 中
    mul ah              ; 执行 8 位乘法, 将 AL*AH 存入 AX
    add edi,eax         ; 将 Y 在 vidbuff 中的偏移地址加到 EDI 上
    add edi,ebx         ; 将 X 在 vidbuff 中的偏移地址加到 EDI 上
    mov al,HBARCHR      ; 将用于生成水平条的字符放入 AL 中
    rep stosb           ; 将这个水平条的字符“爆炸”到缓冲区
    pop edi             ; 恢复我们修改的寄存器
    pop ecx
    pop ebx
    pop eax
    ret                 ; 返回!

;-----------------------------------------------------------------
; Ruler: 在文本缓冲区的 X,Y 处产生一个"1234567890"风格的标尺,
; 最后更新日期: 5/13/2009
; 输入参数: 基于 1 的 X 位置 (行 #)被传递到 EBX 寄存器中
;           基于 1 的 Y 位置 (列 #)被传递到 EAX 寄存器中
;           以字符为单位的标尺的长度被传递到 ECX 寄存器中
; 返回值:无
; 修改: VidBuff
```

```
; 调用：无
; 描述：在视频缓冲区 VidBuff 中由传递到 EBX 和 EAX 的值指定的、
;       基于 1 的 X，Y 位置写入一个标尺，
;       该标尺包含一个数字 1 到 0 组成的重复序列
;       如果该标尺开始的行中盛放不下它，
;       这个标尺将换到后续行并覆盖任何在它的长度范围内的 EOL 字符。
;       注意 Show 过程必须当 Ruler 在控制台上显示标尺后才能被调用

Ruler: push eax              ; 保存我们修改的寄存器
    push ebx
    push ecx
    push edi
    mov edi,VidBuff          ; 加载视频地址到 EDI 寄存器中
    dec eax                  ; 将 Y 的值向下调整 1，用于地址计算
    dec ebx                  ; 将 X 的值向下调整 1，用于地址计算
    mov ah,COLS              ; 将屏幕宽度到移到 AH 中
    mul ah                   ; 执行 8 位乘法，将 AL*AH 存入 AX 中
    add edi,eax              ; 将 Y 在 vidbuff 中的偏移地址加到 EDI 上
    add edi,ebx              ; 将 X 在 vidbuff 中的偏移地址加到 EDI 上
; 现在 EDI 中包含的是缓冲区中的标尺的开始位置的内存地址

; 现在我们从那个位置开始显示标尺：
    mov al,'1'               ; 用数字'1' 开始标尺
    DoChar: stosb            ; 注意，没有 REP 前缀！
    add al,'1'               ; 将 AL 中的字符值增加 1
    aaa                      ; 调整 AX 的值，来完成这个 BCD 加法
    add al,'0'               ; 确保 AL 的高半字节包含二进制 3
    loop DoChar              ; 回去处理下一个字符，一直到 ECX 变成 0
    pop edi                  ; 恢复我们修改的寄存器
    pop ecx
    pop ebx
    pop eax
    ret                      ; 返回！

;------------------------------------------------------------------
; 主程序：

_start:
    nop                      ; 这个无操作指令让 gdb 感到非常高兴......

; 让控制台和文本显示缓冲区做好准备：
    ClearTerminal            ; 发送终端清除字符串到控制台
    call ClrVid              ; 初始化/清除视频缓冲区
```

```
; 下一步，我们显示顶部的标尺：
    mov eax,1               ; 将 Y 的位置加载到 AL 中
    mov ebx,1               ; 将 X 的位置加载到 BL 中
    mov ecx,COLS-1          ; 将标尺的长度加载到寄存器 ECX 中
    call Ruler              ; 将该标尺写到缓冲区中

; 这里我们循环通过数据集并图形化这些数据：
    mov esi,Dataset         ; 把数据集的地址放入 ESI 寄存器中
    mov ebx,1               ; 从左边距(X=1)处开始水平条
    mov ebp,0               ; 数据集元素索引从零开始
.blast: mov eax,ebp         ; 将数据集编号添加到元素索引
    add eax,STRTROW         ; 向下偏置行值
mov cl,byte [esi+ebp]       ; 将数据集的值放入 ECX 寄存器的低位字节
    cmp ecx,0               ; 判断一下是否从数据集中拉出了一个零值
    je .rule2               ; 如果已经从数据集中拉出了一个零,则大功告成
    call WrtHB              ; 将那个数据图形化为水平条
    inc ebp                 ; 增加数据集元素的索引
    jmp .blast              ; 返回，制作另外一个水平条

; 显示底部的标尺：
.rule2: mov eax,ebp         ; 使用数据集计数器来设置标尺的行
    add eax,STRTROW         ; 向下偏置行值
    mov ebx,1               ; 将 X 的位置加载到 BL 中
    mov ecx,COLS-1          ; 将标尺的长度加载到 ECX 寄存器中
    call Ruler              ; 将标尺写到缓冲区中

; 在最后一行的中央显示一个内容丰富的信息
    mov esi,Message         ; 将信息的地址加载到寄存器 ESI 中
    mov ecx,MSGLEN          ; 将信息的长度加载到寄存器 ECX 中
    mov ebx,COLS            ; 将屏幕的宽度加载到寄存器 EBX 中
    sub ebx,ecx             ; 计算信息的长度和屏幕宽度之间的差值
    shr ebx,1               ; 将这个差值被 2 除以求取 X 的值
    mov eax,24              ; 将设置信息所在的行设置为第 24 行
    call WrtLn              ; 在屏幕中央显示信息

; 所有内容已经写入缓冲区后，将缓冲区发送到控制台：
    call Show               ; 用缓冲区刷新控制台
    Exit: mov eax,1         ; 退出系统调用的代码
    mov ebx,0               ; 将返回值设置为零
    int 80H                 ; 进行内核调用
```

11.2 REP STOSB，软件机枪

虚拟文本显示缓冲区只是一块位于.bss 段中的、通过使用 RESB 指令预留的内存区域。缓冲区的大小由两个等值来定义，这两个等值指定想要的行数和列数。默认情况下，我已将其设置为 25 行和 80 列，但是 2010 年的控制台能够显示比它大得多的屏幕。我们可以通过修改 COLS 和 ROWS 等值来将缓冲区定义为 255 × 255 那么大，虽然如果终端窗口没有那么大的话，结果会难以预测。我们可以通过修改一个或两个等值来改变文本显示的尺寸。任何必须对代码做出的其他修改都将自动处理。请注意，这必须在汇编时来做，就像在生成程序时，许多计算必须由 NASM 在汇编时完成一样。

终端窗口的大小不必正好与你所选择的行值和列值精确地匹配，只要比 ROWS×COLS 大即可。如果(像 Konsole 一样)最大化终端窗口，文本将在屏幕的左上角开始显示。

11.2.1 机枪扫射虚拟显示器

当 Linux 将程序加载到内存中时，它通常将未初始化的变量(如清单 11-1 中的 VidBuff)清为二进制零。这是件好事，但是，二进制零并不能在 Linux 控制台上正确地显示。要想看到控制台上“一片空白”，显示缓冲内存必须被清为 ASCII 空格字符。这意味着在内存中写入值 20h，从缓冲区的开始一直到它的结尾。

这种事情应该总是在紧密循环中完成。最明显的办法就是把显示缓冲区的地址放入 EDI 寄存器中，把刷新显示缓冲区的字节数放入 ECX 寄存器中，把用来清除缓冲区的 ASCII 值放入 AL 中，然后通过如下方式编码实现紧密循环：

```
Clear: mov byte [edi],AL ; 将 AL 中的值写入内存
inc edi                  ; 将 EDI 指向缓冲区中的下一个字节
dec ecx                  ; 将 ECX 递减一个位置
jnz Clear                ; 再次循环直到 ECX 中的值变为零
```

这将会工作。它的速度甚至相当快，特别是在较新的 CPU 上，但是上述所有的代码等价于下面的一条单个指令：

```
rep stosb
```

千真万确。

STOSB 指令是最简单的 x86 字符串指令，也是开始讲解字符串相关知识的一个好地方。像上面显示的一样，这条指令包含两个组成部分，我们还从来没有见过这种情况。REP 是一种新型的小家伙，叫作前缀，它改变了 CPU 处理它后面的指令助记符的方式。我们会很快回到 REP 指令。现在，让我们来看看 STOSB 本身。这个助记符的意思是按字节存储字符串。像所有的字符串指令一样，STOSB 使用一些对某些 CPU 寄存器的假设。它仅适用于目标字符串，因此 ESI 寄存器没有参与。但是，我们必须尊重且处理这些假设：

- 必须将把目标字符串的地址加载到 EDI 中。(可以这样认为：E*DI 的 D* 即为目标(*d*estination)的第一个字母，I 即为索引(*i*ndex)的第一个字母。)
- 必须把 AL 中的值即将存入字符串的次数加载到 ECX 中。
- 必须把即将存入字符串中的值加载到 AL 中。

11.2.2 执行 STOSB 指令

一旦设置了这三个寄存器，我们就可以安全地执行 STOSB 指令了。当这样做时，将发生以下事情：

(1) AL 中的字节值被复制到存放在 EDI 中的内存地址中。

(2) EDI 被递增 1，这样，它现在指向内存中刚刚写入的字节的下一个字节。

请注意，这里我们没有扫射——至少还没有开始扫射。AL 的一个副本被复制到内存中的一个位置。EDI 寄存器被进行调整，以便为下一次 STOSB 的执行做好准备。

我们必须记住的非常重要的一点是： STOSB 并不递减 ECX。只有把 rep 前缀放在 STOSB 的前面，ECX 才会自动递减。如果没有 REP 前缀，我们必须自己执行递减操作，或者显式地通过 DEC 指令，或者显式地通过 LOOP 指令，我将在本章稍后的部分讲解这一点。

所以，如果没有 REP 前缀，我们就无法让 STOSB 自动运行。但是，如果喜欢的话，我们可以在执行另一个 STOSB 指令之前执行某些其他指令。只要不“打扰”EDI 或 ECX，我们就可以做任何希望做的事情。然后当再次执行 STOSB 指令时，AL 的另一份副本将会乖乖地走向 EDI 指向的位置，EDI 将会被再次进行调整。(必须要记住以某种方式递减 ECX！) 请注意，如果想的话，我们可以更改 AL 中的值，但是那个更改后的值将被复制到内存中。(我们可能需要

那样做，没有法律规定只能用同一个值填写字符串。在本章后面部分，你将会发现这样做有时非常有用。)

然而，这非常像一个半自动武器(每次按下和释放扳机时，它都射击一发子弹)和一个全自动武器(只要你按下了扳机，它就连续不断地射出一发一发的子弹)之间的差异。为了使 STOSB 指令完全自动化，只需在它的前面挂上 rep 前缀即可。REP 所做的事情非常简洁：它建立了一个所有紧密循环中最紧密的循环，而且完全在 CPU 内部，并且重复地将 AL 中的副本发送到内存(因此得名)，每次将 EDI 递增 1 ，同时将 ECX 递减 1，直到 ECX 递减为 0。然后它就停止，当善后工作处理完毕后，我们会看到整个目标字符串已经添满了 AL 的副本，无论它有多么大。

年轻人，现在，那就是编程！

在清单 11-1 中提供的 vidbuff1 程序中，清空显示缓冲区的代码是在 ClrVid 过程中。相关代码行如下所示：

```
cld                      ; 清零 DF; 我们正在以上内存的方式计数
mov al,FILLCHR           ; 将缓冲区过滤器字符放入 AL 中
mov edi,VidBuff          ; 指向缓冲区中的目标索引
mov ecx,COLS*ROWS        ; 将存储的字符的数量放到 ECX 中
rep stosb                ; 在缓冲区中“爆炸”字符
```

FILLCHR 等值默认情况下被设置为 32，这是 ASCII 空格字符。我们还可以通过将它设置为其他值来为缓冲区填充其他字符，尽管这个能有多大的用处尚不清楚。我们还要注意的是，将被写入内存的字符的数量是由 NASM 在汇编时通过计算 COLS 乘以 ROWS 而得到的。正是因为这一点，我们才可以在不必修改“清空显示缓冲区”的代码的情况下，改变虚拟显示器的大小。

11.2.3　STOSB 和方向标志(DF)

上面显示的简短代码序列的开头是一条我在前面从来没有提到过的指令：CLD。在字符串指令的工作中，它控制着某些关键的东西，那就是字符串操作在内存中采用的方向。

当使用 STOSB 指令时，在大多数情况下我们需要以“上山”的方式在内存中运行——也就是说，从一个较低的内存地址到一个较高的内存地址。在 ClrVid 中，我们把视频刷新缓冲区的起始地址放入 EDI 寄存器中，然后将字符送入后续的更高的内存地址中。每次 STOSB 发送一个字节到内存中之后，EDI 都会递

增，以指向内存中的下一个更高字节。

这是合乎逻辑的工作方式，但是我们不必总是以这种方式来使用它。STOSB 可以轻松地从一个较高的地址开始，然后在内存中向下移动。每次存储一个字符到内存中之后，EDI 都可以被减一所代替。STOSB 采用哪种方式“发射”——沿着后面较高的内存“上山”，还是沿着后面较低的地址“下山”——是由一个位于 EFlags 寄存器中的标志来控制的。这就是方向标志 DF。DF 一生之中的唯一一项工作就是控制某些能够沿着内存中的两个方向之一移动的特定指令(如 STOSB 采取行动的方向。其中大多数(如 STOSB 及其兄弟指令)是字符串指令。

应该这样理解 DF：当 DF 被设置(即，当 DF 的值为 1)时，STOSB 及其兄弟字符串指令以从高地址到低地址的“下山”方式工作。当 DF 被清零(即，当 DF 的值为 0)，STOSB 及其兄弟字符串指令以从低地址到高地址的“上山”方式工作。依次的，这也是 EDI 寄存器被调整的方向：当 DF 被设置时，EDI 在字符串指令执行过程中递减。当 DF 被清零时，EDI 在字符串指令执行过程中递增。

当 CPU 被复位时，方向标志默认为 0(上山方式)。它通常情况下可以通过两种方式进行改变：通过 CLD 指令或者通过 STD 指令。CLD 将 DF 清除为 0，STD 将 DF 设置为 1。(当从堆栈中弹出一套全新的标志到 Eflags 寄存器中，以便对“POPF 指令也能改变 DF”进行调试时，我们应该牢记这一点。) 由于 DF 的默认状态是清 0 的，所以 vidbuff1 演示程序中的所有字符串指令在内存中都以“上山”方式工作，从技术上讲，ClrVid 过程不必非要包含 CLD 指令。然而，程序的其他部分也能够修改 DF，所以在任一字符串指令的前面加上适当的 CLD 指令或者 STD 指令以确保机枪朝着正确的方向射击，总是一个不错的主意！

人们有时会感到困惑，并认为 DF 也控制着 ECX 是被字符串指令递增还是递减。事实并非如此！字符串指令从不递增 ECX 寄存器。ECX 中存放的是一个计数值，而不是一个内存地址。我们将一个计数值放到 ECX 寄存器中，并且当每次字符串指令“射击”时，将计数值周期性地减少，就这样。DF 对此没什么可说的。

11.2.4　在显示缓冲区中定义行

但是，将 VidBuff 清为空格字符并不是故事的最后结局。要想在显示 Linux 控制台的终端程序上正确地渲染，数据必须分行显示。行是由 EOL 字符(ASCII 码 10)分隔的。第一行开始于缓冲区的起始位置，并结束于第一个 EOL 字符。下一行立即从 EOL 字符后面开始，并运行直到下一个 EOL 字符，等等。

当文本被零碎地写入控制台时，每一行的长度都可能有所不同。但是，在我们的虚拟显示系统中，整个缓冲区被作为一系列长度相同的行，通过一条 INT 80h 指令，一次性写到控制台上。这意味着当我们清除缓冲区时，也必须在每个我们希望结束的显示行中插入 EOL 字符。这是在 ClrVid 过程的剩余部分中完成的。我们必须做的事情就是每隔 COLS 个字节将一个 EOL 字符写入缓冲区。这是通过一个非常紧密的循环来实现的。如果观察 ClrVid 的第二部分，我们可能会注意到，其中所涉及的循环都不是很常见。带着这种想法，我将简单地返回 LOOP 指令。

11.2.5　将缓冲区发送到 Linux 控制台

我需要重申一下：这里我们正在谈论的是一个虚拟的显示。VidBuff 仅仅是一个内存区域，我们可以使用普通的汇编语言指令将字符和字符串写入其中。但是，在将缓冲区发送到 Linux 控制台之前，显示器上什么也不会出现。

这很容易做到。清单 11-1 中的 Show 过程通过 INT 80h 对内核服务 sys_write 进行了单次调用，并且一次性将整个缓冲区发送到控制台上。嵌入到缓冲区中的每 COLS 个字节中的 EOL 字符被控制台视为普通的 EOL，并且迫使在每一个 EOL 后面立即开始一个新的行。因为所有行的长度都相同，所以发送 VidBuff 到控制台将创建一个矩形文本区域，而且该区域将会正确显示在大小至少为 COLS 乘以 ROWS 的任何终端窗口中。(较小的窗口将会使 VidBuff 的文本变得混乱。尝试一下在各种大小的终端窗口运行 vidbuff1，你很快就会明白我的意思。)

最重要的是，每当你想更新屏幕时，你的程序都会调用 Show 过程。想多么频繁地调用就多么频繁调用，而且什么时间都可以。在现代的 Linux 个人电脑上，更新发生得如此迅速，以至于每时每刻都会出现。没有理由在每次写入 VidBuff 时不调用 Show 过程，但是这取决于你。

11.3　半自动武器：不带 REP 的 STOSB

在所有的字符串指令中，我选择最先展示 REP STOSB 指令，因为它极为引人注目。但更重要的是，它非常简单——事实上，使用 REP 比不使用 REP 更加简单。从程序员的角度来看，REP 简化了字符串处理，因为它带来了在 CPU 内部循环的指令。我们也可以使用不带 REP 的 STOSB 指令，但是这需要我们多做一点工作。这项工作涉及在 CPU 的外部设置循环指令，并确保指令的正确性。

为什么要自找麻烦呢？原因很简单：如果使用带 REP 前缀的 STOSB，我们只能存储相同的值到目标字符串中。在执行 REP STOSB 之前放入 AL 中的任何值都将被发射到内存中 ECX 次。通过在触发扳机之间改变 AL 的值来进行“半自动”发射，STOSB 也可用于将不同的值存储到目标字符串中。

我们会在自己处理 CPU 外面的循环上浪费一点时间，因为必须花费一些时间从内存中获取循环指令字节。不过，如果能够让循环尽可能紧密，速度不会降低到令人反感的程度，特别是在现代处理器上，我们可以非常有效地利用缓存，因而不必每次执行时都从内存中取指。

11.3.1　是谁递减了 ECX

在我的早期汇编语言经验中，我记得感到最为糊涂的就是当使用字符串指令时，ECX 寄存器在哪里递减以及何时被递减。这是一个关键性问题，尤其是在不使用 REP 前缀时。

当使用 REP STOSB(或任何带有 REP 前缀的字符串指令)时，在每个内存访问指令发生时，ECX 都被自动递减 1。而且一旦 ECX 将自身递减到 0，REP STOSB 将检测到 ECX 的当前值为 0，进而会停止对内存的射击。然后控制被传递给下一条排队的指令。但是如果拿走 REP，ECX 将停止自动递减。因而对于“ECX 中的计数值是否已经减少到 0”的自动检测也会停止。

显然，必须有某些东西来递减 ECX，因为 ECX 控制着字符串指令访问内存的次数。如果 STOSB 不这样做——你猜对了——我们必须在其他地方通过使用另一条指令这样做。

递减 ECX 的最明显的方法是使用 DEC ECX，判断 ECX 是否已经递减为 0 的最明显的方式是在 DEC ECX 指令后面加上一条 JNZ(如果不为零跳转)指令。JNZ 测试零标志，ZF，并且跳回到 STOSB 指令，直到 ZF 变为真。当一个 DEC 指令使其操作数(在这里，ECX)变为 0 时，ZF 变为真。

11.3.2　LOOP 指令

了解了上述所有内容之后，请思考下面的汇编语言指令循环。请注意，我已通过插入两个空行将其分成三个部分：

```
DoChar: stosb        ; 注意，已经没有了 REP 前缀！
    add al,'1'       ; 将 AL 中的字符值加 1
```

```
    aaa             ; 调整 AX 使之成为一个 BCD 加法
    add al,'0'      ; 从本质上来讲，将二进制值 3 放入 AL 的高半字节

    dec ecx         ; 将计数器递减 1
    jnz DoChar      ; 如果 ECX > 0，继续循环
```

暂时忽略中间的那个 3 条指令的代码块。这三条指令所做的就是我在前面不远处建议的事情：在每次存储 AL 到内存之间改变 AL 中的值。我将会稍后详细解释这是如何实现的。而(现在)我们要看看循环是如何运行的。STOSB 发射，AL 被修改，然后 ECX 被递减。JNZ 指令执行测试，看看是否 DEC 指令迫使 ECX 变成了零。如果是这样，零标志 ZF 被设置，并且循环终止。但是，在 ZF 被设置之前，转移将跳回标签 DoChar，在那里 STOSB 开始再次射击。

有一种简单的方法，它使用一条至今为止我还没有讨论过的指令：LOOP(循环指令)。LOOP 指令将递减 ECX 和一个测试相结合，并且基于 ZF 的值进行跳转。它看起来像这样：

```
DoChar: stosb       ; 注意，已经没有了 REP 前缀！
    add al,'1'      ; 将 AL 中的字符的值加 1
    aaa             ; 调整 AX 使其成为一个 BCD 加法
    add al,'0'      ; 确保我们已经把二进制值 3 放入了 AL 的高半字节
    loop DoChar     ; 回去继续处理另一个字符，直到 ECX 变为 0
```

LOOP 指令首先将 ECX 递减 1。然后它检查零标志，看看该递减操作是否迫使 ECX 变为零。如果是这样，它往下走，进入下一条指令。如果不是(即，如果 ZF 仍然 0，表明是 ECX 仍然大于 0)，则循环进入其操作数指定的标号这一分支。

循环不断地循环执行 LOOP 指令，直到 ECX 计数值递减为 0。此时，循环结束，执行继续进行到 LOOP 指令后面的那条指令。

11.3.3　在屏幕上显示一个标尺

为了演示如何使用不带 REP 的 STOSB(但是使用 LOOP)指令，让我们为视频工具包提供另一个项目。清单 11-1 中的 Ruler 过程在屏幕上某一可选位置上显示一个从 1 开始递增的、数字的、可以为任意长度的重复序列。换句话说，你可以在任意你想要的地方显示这样一个数字字符串：

```
123456789012345678901234567890123456789012345678901234567890
```

你可以决定控制台窗口水平方向的某一行或者某些字符从何处开始显示。

Ruler 过程使你能够(以数字为单位)指定标尺的长度，以及在屏幕上显示的位置。

一个典型的对于 Ruler 过程的调用看起来像这样：

```
mov eax,1                ; 加载 Y 的位置到 AL 中
mov ebx,1                ; 加载 X 的位置到 BL 中
mov ecx,COLS-1           ; 加载标尺的长度到 ECX 中
call Ruler               ; 将标尺写入缓冲区中
```

该调用将在显示器的左上角位置(1,1)处开始放置一个标尺。标尺的长度通过 ECX 寄存器传递。这里指定的标尺长度比显示器(显示区域)的宽度少一个字符。它提供了一个横跨虚拟文本显示器的整个可见宽度的标尺。

为什么要少一个字符呢？请记住，在每一行的末尾都有一个 EOL 字符。这个 EOL 字符不是直接可见的，但是它仍然是一个字符，因此必须在缓冲区中留出一个字节来存放它。COLS 等值必需始终考虑到这一点：如果想要一个 80 个字符宽的显示屏，COLS 必须被设置为 81。如果想要一个 96 个字符宽的显示屏，COLS 必须被设置为 97。如果你编写了如上所示的代码以实现对 Ruler 过程的调用，NASM 将会做一些编译时数学运算，并且总是产生一个横跨文本显示器全部(可见)宽度的标尺。

在 LOOP 指令之上，有很多新的汇编技术在工作，这将需要很多的解释。让我们暂时先绕过字符串指令，仔细看一看。

11.3.4　MUL 并非 IMUL

我已经在第 7 章中描述了 MUL(乘法)指令及其隐式操作数。Ruler 过程也使用了 MUL 指令，以计算显示缓冲区中的一个 STOSB 可以开始放置标尺字符的 X，Y 位置。用来确定任一给定的 X Y 值在缓冲区中的、以字节为单位的偏移地址的算法看起来是这样的：

```
Offset = ((Y * 以字符为单位的显示器的一行的宽度) + X)
```

显然，我们必须在屏幕缓冲区中向下移动 Y 行，然后再从屏幕的左边距处(向右)移动 X 个字节才能到达(X，Y)位置。在 Ruler 过程中，计算是以这种方式来完成的：

```
mov edi,VidBuff      ; 加载视频地址到 EDI 中
dec eax              ; 将 Y 的值往下调整 1，用于做地址计算
dec ebx              ; 将 X 的值往下调整 1，用于做地址计算
```

```
mov ah,COLS          ; 将屏幕的宽度放入 AH 中
mul ah               ; 执行 8 位乘法：AL*AH ，并将结果存回 AX 寄存器中
add edi,eax          ; 将 Y 在 vidbuff 中的偏移地址放入 EDI 寄存器中
add edi,ebx          ; 将 X 在 vidbuff 中的偏移地址放入 EDI 寄存器中
```

两个 DEC 指令“照顾”了(X，Y)位置在这个系统中基于 1 这一事实，即屏幕的左上角的位置是(1,1)，而不是(0,0)，像它们在某些其他(X，Y)坐标系统中一样。可以这样来理解它：如果想从屏幕的最左上角开始显示一个标尺，必须从缓冲区的最起始位置开始写入标尺字符，没有任何偏移地址。那么，为了计算考虑，(X，Y)的值要必须基于 0。

对于使用 MUL 指令的 8 位乘法而言，其中一个因子是隐式的：AL 包含 Y 的值，主调程序传递给 Ruler 过程的 Y 值存放到寄存器 EAX 中。我们将屏幕的宽度值放入 AH 中，然后将 AH 与 AL 通过使用 MUL 指令进行相乘操作。(如果在此期间感到有些模糊的话，请参见第七章中关于 MUL 的讨论。) 计算结果会取代 AH 和 AL 中的值，并作为 AX 寄存器的值而被访问。将那个结果和(通过寄存器 BL 传递给 Ruler 过程的)X 的值与 EDI 寄存器中的值相加，将得到标尺字符必须被写入的精确内存地址。

现在，有一个相当普遍的 bug，我要在这里提醒你：MUL 不是 IMUL——大部分时间是这样的。MUL 和 IMUL 是都能够执行乘法的“姐妹”指令。MUL 将其操作数作为无符号值来对待，而 IMUL 则将其操作数作为有符号值来对待。如果两个因子在一个有符号上下文环境中一直保持全为正数，这种差异并不重要。在给定 8 位乘法的实际条件下，MUL 和 IMUL 在 127 及小于该值的操作数上以相同的方式工作。但是当操作数为 128 时，情况将发生彻底的变化。在一个 8 位有符号上下文中，127 以上的值被认为是负数。MUL 认为 128 是……128。而 IMUL 则认为 128 是-1。老天！

在 Ruler 过程中，我们可以将 MUL 指令替换为 IMUL 指令，并且该过程将同等地进行工作，一直到传递给它的屏幕尺寸大于 127。然后，突然间，IMUL 会计算出一个有名无实的负数结果……但是只有当将其当作为有符号值的时候才是这样。如果负数被当作无符号数处理，将是一个非常大的正数，EDI 所代表的内存地址引用和这个巨大的值相加，将会产生一个分段错误。试试吧！不会带来任何伤害，它只是一个有趣的教训。IMUL 是为有符号值而准备的。对于内存地址计算而言，不要管它，一定要确保使用 MUL 来代替它。

11.3.5 添加 ASCII 数字

一旦正确计算出标尺的开始位置在缓冲区中的偏移地址并将其放到 EDI(一旦我们设置了 ECX 和 EAX 的初始值)中，我们就准备好开始制作标尺了。

紧挨在 STOSB 指令之前，我们将 ASCII 数字“1”加载到 AL 中。请注意，在指令 MOV AL，‘1’中，这个‘1’并不是指将阿拉伯数字 1 的二进制值移到 AL 中！这个‘1’是一个 ASCII 字符(因为它位于单引号内)并且这个字符‘1’(即数元‘1’)的数值为十六进制的 31h，或十进制的 49。

当通过 STOSB 指令将数元‘1’存储到视频内存(即显存)后，马上带来了一个问题。在数元‘1’之后，我们需要显示数元‘2’——并且为了做到这一点，我们需要将存储在 AL 中的值从‘1’改为‘2’。

通常情况下，我们不能只是简单地将‘1’和‘1’相加以获得‘2’。31h + 31h 会得到 62h，这(当看作一个 ASCII 字符时)是小写字母 b，而不是‘2’！但是，在这种情况下，x86 指令集可以前来救援，我们可以使用一条名为 AAA 的、有点特殊的指令，在 BCD 码加法后对 AL 进行调整。

AAA 所做的事情实际上就是将 ASCII 字符数字而不是数值数字相加。AAA 是 BCD 指令组中的一员，之所以叫这个名字是因为它们支持二—十进制编码(Binary Coded Decimal)，简称为 BCD 码，运算。BCD 码只是另外一种表达一个数值的方法，有点介于纯粹的二进制值(如 1)和 ASCII 码数字(如‘1’)之间。BCD 码值是一个占用一个字节的低半字节的 4 位值。它只能表达介于 0 和 9 之间的值。(这就是二—十进制编码的十进制部分所代表的内容。)使用 4 位有可能表达大于 9(实际上，从 10 到 15)的值，但这些额外的值是无效的 BCD 码值(见图 11-1)。

31h 是一个有效的 BCD 码值，因为它的低半字节包含 1。BCD 码是一个 4 位的编码系统，高半字节(在 31h 这个例子中，包含数字 3)会被忽略。事实上，所有从‘0’到‘9’的 ASCII 数字都可被视为合法的 BCD 码值，因为在每个字符的低 4 位都包含有效的 BCD 码值。在每个 ASCII 数字的高 4 位存储的数字 3 会被忽略。

所以，如果有一种方法可在 x86 处理器上执行 BCD 加法，将‘1’和‘1’相加确实会带给我们‘2’，因为‘1’和‘2’都可被作为合法的 BCD 码值来操作。

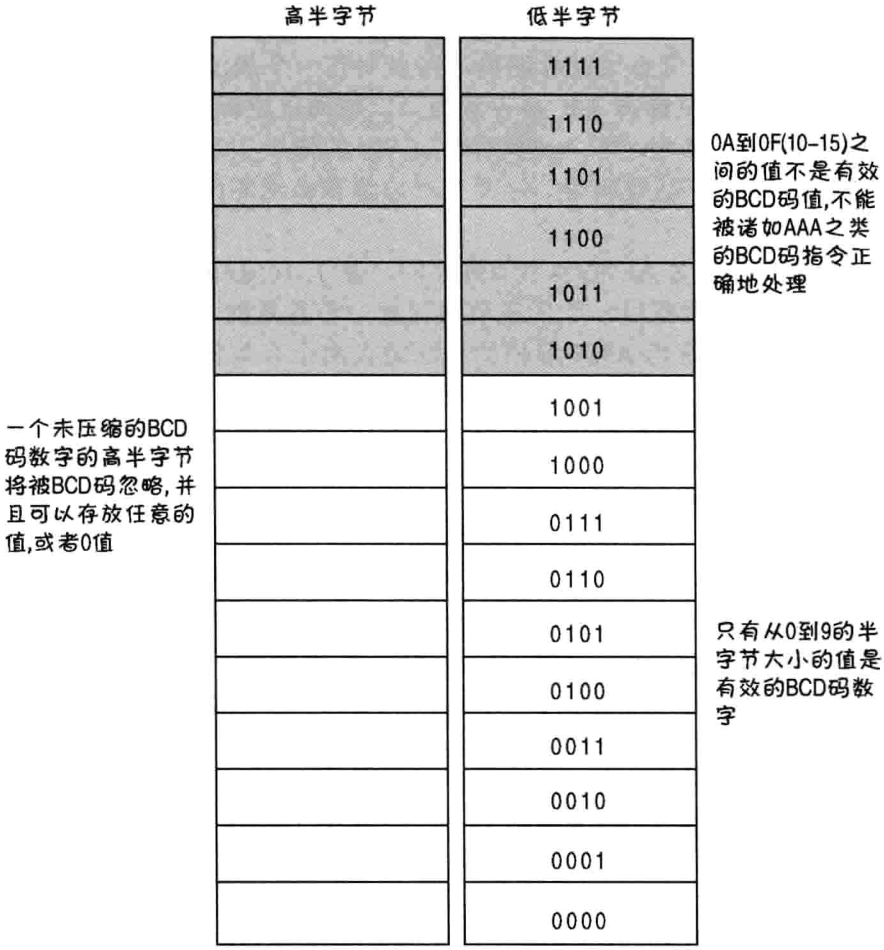


图 11-1　未压缩的 BCD 码数字

AAA(和一些我在本书中没有空间讨论的其他指令)使我们能够执行 BCD 码数学运算。它所使用的实际技术可能看起来有点奇怪，但它确实能够工作。AAA 实际上是一种附加因素，因为 AAA 在使用正常的加法指令 ADD 执行一个加法之后才执行。 AAA 注意到 ADD 指令的结果，并迫使它们按照 BCD 码算术出现。

从根本上来讲，AAA 能够做以下两件事情：

- 如果 AL 中的低 4 位值(可以是任何值 0 到 F 之间的值)大于 9，它将其强制转换为 0 到 9 之间的值。这是通过将 AL 与 6 相加，然后迫使高半字节变为 0 来实现的。显然，如果 AL 的低半字节包含一个有效的 BCD 码

数字，那么这低半字节中的数字不需做任何处理。

- 如果 AL 中的值必须进行调整，则表明有一个加法进位，因此，AH 递增 1。此外，进位标志 CF 被设置为 1，辅助进位标志 AF 也被设置。同样地，如果当 AAA 指令被执行时，AL 的低半字节包含一个有效的 BCD 码，那么 AH 不被递增，这两个标志位都被清零(强制为 0)，而不是被设置为 1。

因此，AAA 可使对 AL 低半字节进行的、基于 10(decimal)的加法更为便利。当 AL 被 AAA 指令调整后，低半字节将包含一个有效的 BCD，而高半字节则包含 0。(但是注意，只有当 AAA 前面的加法是在两个有效的 BCD 码操作数上执行的，这才为真——并且确保这些操作数是有效的 BCD 是你的而不是 CPU 的责任！) 基于以上描述，我们可以使用 ADD 指令来将诸如 1 和 2 之类的 ASCII 数元相加。Ruler 过程在 STOSW 指令之后立即做了这件事情：

```
add al,'1'          ; 将 AL 中的字符值增大 1
aaa                 ; 调整 AX 使之成为一个 BCD 码加法
```

如果在加法之前，AL 的低半字节的内容为 9，将其与 1 相加将会产生一个 0AH 值，这不是合法的 BCD 码值。AAA 将会通过将其与 6 相加并清除高半字节的内容来调整 AL 的值。将 0A 与 6 相加将会产生结果 10，所以一旦高半字节被清零，AL 中的新值将是 0。另外，AH 将被递增 1。

在 Ruler 过程中，我们没有添加多个十进制列，只涉及一个单个的计数值列，并将该列中的数字显示到屏幕上。因此，我们不理会增加后的 AH 中的值，只是单独使用 AL。

11.3.6 调整 AAA

这里有一个问题：AAA 将高半字节清除为 0。这意味着将‘1’和‘1’相加不完全等于可显示数字‘2’。相反的，AL 变成了二进制 2，在 IBM - 850 字符集中是黑色的“笑脸符”字符。为了使 AL 中的内容再次成为一个可显示的 ASCII 数字，我们需要将 AL 中的值与 30h 相加。这很容易实现：只需将 0 与 AL 相加即可(0 是一个值为 30h 数字值)。因此，与 0 相加使得 02h 变成了 32h，这与 ASCII 字符 2 的十进制数相同。这就是紧跟在 AAA 指令之后有一条“ADD AL,‘0’”指令的原因。这听起来有点奇怪，但是请记住，'0'是数字 30h，不是二进制 0！

关于 BCD 码数学运算还有很多内容。其中大部分涉及多列 BCD 码操作。例

如，当执行多列 BCD 码数学运算时，你必须考虑进位，这涉及小心使用辅助进位标志 AF。还有 AAD、AAM、AAS 指令分别用于在 BCD 码除法、BCD 码乘法、BCD 码减法后调整 AL 的值。同样的一般思想依然适用：所有的 BCD 码调整指令都强制标准的二进制算术运算指令同意支持有效的 BCD 码操作数。

11.3.7　Ruler 过程的教训

Ruler 过程是一个使用不带 REP 前缀的 STOSB 指令的很好示例。每次将 AL 中的值存储到内存中时，我们都必须修改 AL 中的值，因此不能使用带 REP 的 STOSB 指令。请注意，在修改即将显示的数字的过程中，我们不对 EDI 或者 ECX 做任何事情，因此在这些寄存器中的值会一直为 STOSB 的下一次执行保留。Ruler 过程也是一个关于 LOOP 指令如何与 STOSB 共同工作以实现向下调整 ECX 和将控制返回到循环顶部的很好示例。从某种意义上来讲，LOOP 指令在 CPU 之外完成 REP 在 CPU 之内所做的事情：调整 ECX 的值和关闭循环。在使用任何字符串指令时，尽量把它的工作原理搞清楚！

11.3.8　STOS 指令的 16 位版本和 32 位版本

在继续讲解其他字符串指令之前，值得一提的是有三种不同“尺寸”的 STOS 字符串指令：字节、字和双字。 STOSB 是 Ruler 过程已经演示过的字节大小的版本。STOSW 将存放在 AX 中的 16 位值存储到内存中，STOSD 将存放在 EAX 中的 32 位值存储到内存中。

STOSW 和 STOSD 的工作方式几乎与 STOSB 一样相同。主要的不同之处在于每次内存传输操作之后对 EDI 的修改方式。对于 STOSW 指令而言，EDI 被改变两个字节，可能向上或者向下，取决于 DF 的状态。对于 STOSD 指令而言，EDI 被改变四个字节，再次地，向上或者向下，取决于 DF 的状态。

但是，在任何情况下，只要指令的前面带有 REP 前缀，ECX 就在每次内存传输操作之后被递减 1。它总是被递减，而且总是被递减 1。ECX 用于对操作进行计数。它与内存地址毫不相干。

11.4　MOVSB：快速块拷贝

STOSB 是一条令人着迷的指令，但是对于包装在一个单行汇编代码中的纯粹

行动而言，没有什么指令能够比得过 MOVS 指令。像 STOS 一样，MOVS 也有三种“尺寸”：处理字节(MOVSB)、处理 16 位字(MOVSW)和处理 32 位双字(MOVSD)。对于处理 ASCII 字符的相关工作，比如同本章所做的事情，我们即将用到的 MOVS 指令是 MOVSB。

MOVSB 指令的要点是：一块存放在 ESI 指定的内存地址中的内存数据被复制到存放在 EDI 寄存器中的内存地址里。即将被移动的字节数存放在 ECX 寄存器中。每当一个字节被复制之后，ECX 中的计数值递减 1，而且将 ESI 和 EDI 中的地址调整 1。对于 MOVSW 而言，每当一个字被复制之后，ESI/EDI 地址被调整 2，对于 MOVSD 而言，每当一个双字被复制以后，ESI/EDI 地址被调整 4。这些调整不是递增就是递减，取决于 DF 的状态。在所有三种情况下，每当一个数据项从源地址去往目的地址，ECX 中的值都会递减一。请记住，ECX 是对内存传输操作计数，而不是对地址字节进行计数。

DF 寄存器影响 MOVSB 的方式与它影响 STOSB 的方式相同。默认情况下，DF 被清除为 0，字符串操作采用从低内存向高内存的“上山”方式进行工作。如果 DF 被设置为 1，那么字符串操作以相反的方向从低内存向高内存操作。如同 STOSB 一样，MOVSB 可以以自动或者半自动的方式进行工作。如果在 MOVSB 指令前面加上 REP 前缀，那么(假设寄存器设置正确)仅通过一条指令，我们就可以实现将一块内存中的数据从这里复制到那里。

为了演示 MOVSB，我向清单 11-1 中添加了一个名为 WrtLn 的简短过程。WrtLn 将一个字符串复制到显示缓冲区 VidBuff 的某一给定的(X，Y)位置。它做的工作很像 Pascal 中的 Write 或者 C 语言中的 print。在调用 WrtLn 过程之前，我们将字符串的源地址放入 ESI 中，基于 1 的 X，Y 坐标分别放入 EBX 寄存器和 EAX 寄存器中，并且将字符串的以字节为单位的长度放入 ECX 寄存器中。在 WrtLn 中完成这项工作的代码相当简单：

```
cld                    ; 清除 DF，以便进行上内存写入工作
mov edi,VidBuff        ; 将缓冲区地址作为目标索引加载到 EDI 寄存器中
dec eax                ; 将 Y 的值向下调整 1，用于地址计算
dec ebx                ; 将 X 的值向下调整 1，用于地址计算
mov ah,COLS            ; 将显示器的宽度放入 AH 中
mul ah                 ; 执行八位乘法，AL*AH，将结果放入 AX 寄存器中
add edi,eax            ; 将 Y 在 vidbuff 中的偏移地址放入 EDI 寄存器中
add edi,ebx            ; 将 X 在 vidbuff 中的偏移地址放入 EDI 寄存器中
rep movsb              ; 将这个字符串“爆炸”到缓冲区中
```

使用 MUL 指令根据(X，Y)的值计算在 VidBuff 中的偏移地址的代码与在 Ruler 过程中的是一样的。在 vidbuff1 的主程序部分，需要执行一些额外的计算，以实现在任一可显示的缓冲区的中央而不是在某一特定的(X，Y)位置显示一个字符串：

```
mov esi,Message     ; 将消息的地址加载到寄存器 ESI 中
mov ecx,MSGLEN      ; 将消息的长度加载到 ECX 寄存器中
mov ebx,COLS        ; 将屏幕的宽度加载到 EBX 寄存器中
sub ebx,ecx         ; 计算消息的长度和屏幕的宽度之间的差值
shr ebx,1           ; 将该差值除以 2，以求得 X 的值
mov eax,24          ; 将消息的行设置为第 24 行
call WrtLn          ; 显示这个居中的消息
```

11.4.1　DF 和重叠块移动

简单的演示程序 vidbuff1 使用 MOVSB 指令从程序的.data 段复制一条消息到缓冲区中。WrtLn 采用“上山”的方式使用 MOVSB 指令从低内存复制一条消息到高内存，你可能认为能够同样容易地以“下山”的方式将其从高内存复制到低内存，你是正确的。方向标志 DF 看起来似乎只是一个个人喜好的问题……直到源内存块和目标内存块重叠。没有人规定 ESI 和 EDI 必须指向完全独立的内存区域。源内存块和目标内存块可能会重叠，而且那种情况往往是非常有用的。

这里有一个例子：思考一下编辑存储在内存缓冲区中的文本这一挑战。假设在缓冲区中有一个字符串，并且现在你想将一个字符插入到字符串的中间某处。所有超过插入点的字符必须被“移开”以便为新的插入字符腾出空间(假设在缓冲区末尾处是空白的)。这是一个很自然的使用 REP MOVSB 的应用——但是设置起来可能比乍看之下要复杂一些。

我清楚地记得我第一次尝试它的情景——并非巧合，那是我第一次尝试使用 MOVSB。我所做的事情已经以示意图的方式显示在图 11-2 的左半部分。目标是将字符串向右移动一个位置，以便可以在它的前面插入空格字符。(在当时，我使用的是 16 位的 CPU，寄存器为 SI、DI 和 CX，但是机制与 32 位模式精确地相同。只是寄存器的名称与今天的不同。)

我让 ESI 指向字符串的第一个字节，让 EDI 指向我想将字符串移到的位置。然后执行一个“上山”方式的 REP MOVSB 指令，当执行完毕后，我发现已经将整个字符串用它的起始字符取代了。是的，这是一个很明显的错误……一旦你看到它实际发生。

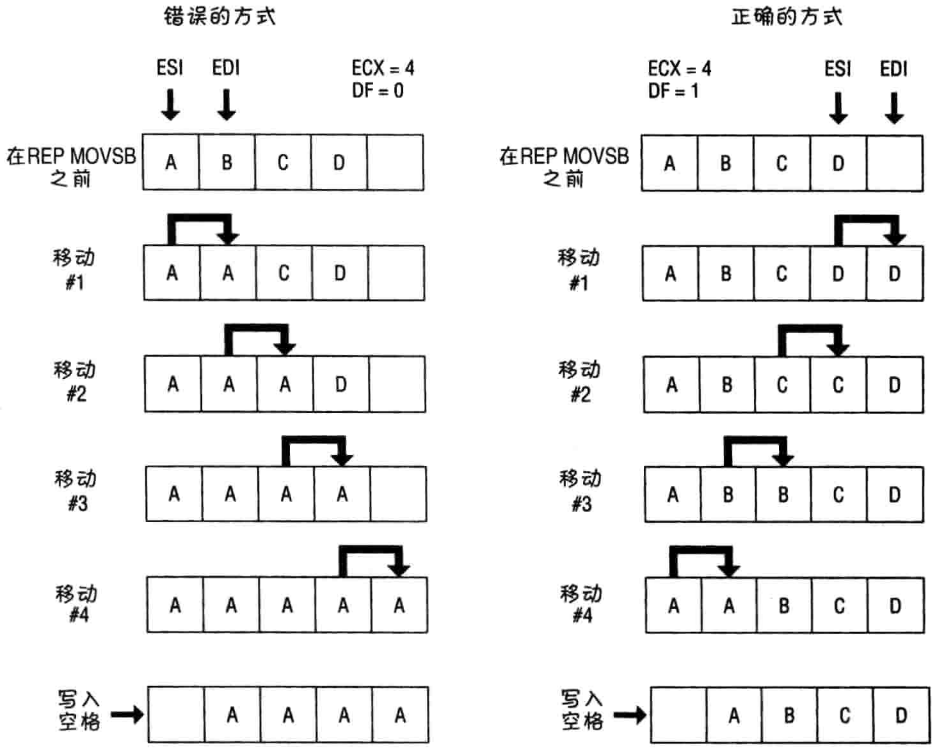


图 11-2　在重叠内存块儿上使用 MOVSB

该图的右半部分显示了这样的插入操作实际上应该如何完成。我们必须从字符串的结尾处开始，然后以“下山”的方式朝着插入点的方向开始工作。第一次字符移动必须将字符串的最后一个字符放入缓冲区的空白区域，为下一个字符的移动让路，等等。这样，两块只有一个字节没有重叠的内存区域才能够从一个复制到另外一个上，不会丢失任何数据。

观察类似这样的操作的发生情况非常容易，通过在一个沙盒程序中设立一个测试案例，并在 Insight 的存储器视图中对内存进行观察即可。输入如下所示的沙盒程序，生成可执行文件，然后把它放到 Insight 中：

```
section .data
    EditBuff: db 'abcdefghijklm '
    ENDPOS equ 12
    INSRTPOS equ 5
section .text
```

```
    global _start
_start:
    nop
; 将你的实验放到两个 nop 指令之间
    Std                             ; 采用下内存方式传输
    mov ebx,EditBuff+INSRTPOS       ; 保存插入点的地址
    mov esi,EditBuff+ENDPOS         ; 从文本的结尾开始
    mov edi,EditBuff+ENDPOS+1       ; 将文本向右移动 1 个位置
    mov ecx,ENDPOS-INSRTPOS+1       ; 打算移动的字符的个数
    rep movsb                       ; 移动它们
    mov byte [ebx],' '              ; 在插入点写入一个空格
; 将你的实验放到两个 nop 指令之间
    nop
```

确保 Insight 的存储器视图窗口处于打开状态。字符串 EditBuff 将被显示在存储器视图的顶部。通过单步调试沙盒代码，你可以一个接一个地观察字符如何在 EditBuff 变量内移动。

在这个例子中，ENDPOS 是最后一个字符在字符串中的、以零为起点的偏移地址。请注意，这不是一个计数值，而是相对于缓冲区开始位置的偏移地址。最后的字符“m”相对于缓冲区的开始位置的偏移地址是 12 个字节。如果从存放在 ESI 中的 EditBuff 的地址开始，那么将 12 与 ESI 相加，ESI 将指向“m”。依次地，EDI 指向字符串最后一个字符后面的第一个缓冲区位置的偏移地址；因此，有 ENDPOS +1 编译时计算。

推导即将放入 ECX 寄存器中的计数值必须要考虑到偏移地址基于零这一特性。我们必须将字符串的结束位置(ENDPOS)和插入点位置(INSRTPOS)之间的差值加 1 才能获得必须移动的字节数的正确计数。

请注意代码块开头的 STD 指令。STD 将方向标志 DF 设置为 1，这强迫字符串采用从高内存到低内存的“下山”方式工作。DF 的默认值为 0，为了使这里的代码能够正常工作，STD 指令必须在场！

我们还可以在沙盒示例程序中改变插入点的位置，只需改变 INSRTPOS 的值，然后再重新生成这个沙盒程序即可。例如，如果要在字符串的最开始位置插入空格字符，将 INSRTPOS 的值改为 0 就可以了。

11.4.2　使用 Insight 单步调试 REP 字符串指令

我应该在这里提一下，虽然 REP MOVSB 指令看起来是一个单指令，但它实

际上是一个作为单指令实施的非常紧密的循环。在 Insight 下单步调试 REP MOVSB 并不能一举执行整个循环！每次单击“单步 ASM 指令”按钮时，只有一个内存传输操作发生。例如，如果 ECX 寄存器被加载的计数值为 12，那么你必须单击“单步 ASM 指令”按钮 12 次，才能单步走完整个指令。

有时这是一件好事，特别是当我们想要观看这条指令操作时的内存变化时。不过，如果在 ECX 寄存器中的计数值较大，那么这将是一个很大的点击次数。如果对字符串指令设置的正确性有信心，我们完全可以在 REP 字符串指令的下一条指令处设置一个断点，然后点击“继续”按钮全速执行，而不必每次在内存传输操作后暂停。Insight 将在断点处暂停，我们可以从那里开始继续进行单步调试。

11.5 将数据存储到不连续的字符串中

有时你必须打破规则。到现在为止，我对于字符串指令的解释都是基于这样的假设：目标字符串始终是内存中的一个连续字节序列。但是情况并非一定如此。除了可以在 STOSB 指令执行期间更改存放在 EAX 寄存器中的值之外，我们还可以更改目标地址。因此，可以在一个非常紧密的循环内将数据存储到不同的内存区域。

11.5.1 显示一个 ASCII 表

我已经创建了一个名为 showchar 的小演示程序，以展示我所说的含义。它不像清单 11-1 中的 Ruler 过程一样有用，但是也有其令人欣赏的地方，而且到目前为止，如果你一直跟着我学习的话，很容易理解它。因为 showchar 过程使用了很多与 vidbuff1 相同的基础方法，包括虚拟显示机制和标尺，我不打算在这里显示整个程序。完整的源代码文件(与所有本书介绍的代码)可以直接从我的汇编语言网页上下载，在清单存档 zip 文件中。(网址已被打印在本书的简介中。)

showchar 程序将屏幕清空，在第 1 行显示一个标尺，在标尺下面显示一个表，其中包含 256 个 ASCII 字符中的 224 字符，整齐地显示在 7 行中，每行包括 32 个字符。本表包括“高”127 个 ASCII 字符，包括外国语言字符、画线字符和其他符号。它不显示最前面的 32 个 ASCII 字符。Linux 将这些当作控制字符，即使这些字符对应的字形可以得到，它们也不会被显示到控制台上。

showchar 程序介绍了一对儿新的概念和指令，它们都与程序循环有关。(与

诸如 STOSB 之类的字符串指令和程序循环密切相关。）清单 11-2 给出了 showchar 程序的主体部分。所有它所调用的过程和宏都展示在清单 11-1 中。它还使用了以下两个在清单 11-1 中没有出现的等值：

```
CHRTROW equ 2       ; 图表从距离显示器顶部 2 行的位置开始
CHRTLEN equ 32      ; 每个图表行显示 32 个字符
```

在继续正文的学习之前，请仔细阅读以下代码。

清单 11-2：showchar.asm (只包括程序主体部分)

```
_start:
    nop                     ; 这个无操作指令让 gdb 感到十分高兴

; 让控制台和文本显示的文本缓冲区做好准备：
    ClearTerminal           ; 将终端清除字符串发送到控制台
    call ClrVid             ; 初始化/清除视频显示缓冲区

; 在表格显示区域的上方显示一个 64 个字符宽的标尺：
    mov eax,1               ; 从位置(1, 1)处开始显示标尺
    mov ebx,1
    mov ecx,32              ; 将标尺的宽度设置为 32 个字符
    call Ruler              ; 生成这个标尺

; 现在让我们生成图表本身：
    mov edi,VidBuff         ; 从 ESI 中存放的缓冲区地址开始
    add edi,COLS*CHRTROW    ; 在 CHRTROW 行下面开始显示表
    mov ecx,224             ; 显示 256 个字符除去前 32 个之后的内容
    mov al,32               ; 从第 32 个字符开始显示；其他的不会显示
.DoLn: mov bl,CHRTLEN       ; 每行将包括 32 个字符
.DoChr: stosb               ; 注意，没有 REP 前缀！
    jcxz AllDone            ; 当整个序列被打印完毕，退出
    inc al                  ; 将 AL 中的字符的值增大 1
    dec bl                  ; 将计数器的值递减 1
    loopnz .DoChr           ; 回去并处理另外一个字符直到 BL 变为 0
    add edi,(COLS-CHRTLEN)  ; 将 EDI 挪到下一行的开始位置
    jmp .DoLn               ; 开始显示下一行

; 已经把所有内容写入缓冲区后，将缓冲区发送到控制台：
AllDone:
    call Show               ; 使用缓冲区刷新控制台
```

```
Exit: mov eax,1          ; 退出系统调用的代码
      mov ebx,0          ; 以零作为返回值而返回
      int 80H            ; 进行内核调用
```

11.5.2　嵌套指令循环

一旦所有寄存器都根据 STOSB 所做的假设正确设置之后，showchar 的实际工作由两个指令循环来实现，一个指令循环在另一个指令循环的内部。内层循环显示一个由 32 个字符组成的行。外层循环将显示分解成 7 个这样的行。这里显示的内层循环是两个循环中比较有趣的：

```
.DoChr: stosb        ; 注意这里没有 REP 前缀！
jcxz AllDone         ; 当所有序列都打印出来后，退出
inc al               ; 将 AL 中的字符值增加 1
dec bl               ; 将行计数器递减 1
loopnz .DoChr        ; 返回并处理另外一个字符，直到 BL 变为 0
```

这里的工作(把一个字符放到显示缓冲区中)又是由 STOSB 指令来完成。再一次地，STOSB 正在独唱，没有 REP 伴奏。如果没有 REP 将循环拉到 CPU 内部，那么我们必须自己设置循环。

请记住每次 STOSB“开火”时都发生什么：在 AL 中的字符被写入 EDI 寄存器所指向的内存位置，EDI 递增 1。在循环的另一端，LOOPNZ 指令将 ECX 递减 1，并关闭循环。

在寄存器的设置过程中，我们把将要显示的字符的个数加载到 ECX 寄存器中，在本例中为 224。STOSB 每次“开火”时，都将另一个字符放入显示缓冲区 VidBuff 中，并在左侧少显示一个字符。ECX 充当主计数器使用，它跟踪我们什么时候最终显示最后的剩余字符。当 ECX 变为零时，我们已经显示了 ASCII 字符集的一个适当子集，所有任务完成。

11.5.3　当 ECX 变为 0 时进行跳转

因此，我们再来看看 JCXZ 指令。这是一条专门为帮助类似这样的循环而创建的特殊的分支指令。在第 10 章中，我解释了如何使用 JMP 指令的众多变体基于计算机的某个标志的状态实现可能的分支。在本章的前面部分，我解释了 LOOP 指令，它是 JMP 指令的一种特殊用途，一个与隐式的 DEC ECX 指令合在一起的用途。JCXZ 又是另外一种 JMP 指令，但是它是一条不观察任何标志，或

递减任何寄存器的指令。相反，JCXZ 观察 ECX 寄存器。当看到 ECX 变为零的时候，它跳转到指定的标签处。如果 ECX 仍然是非零，那么执行失败，直接进入下一条指令。

在前面展示的内部循环中，当看到 ECX 最终变为 0 时，JCXZ 跳转到“关店”代码分支。showchar 程序就是这样终止的。

大多数 JMP 指令有一些当控制标志不为真时进行跳转的搭档。也就是说，当进位标志等于 1 时，JC(当进位时则跳转)进行分支跳转。它的搭档，JNC(当没有进位时则跳转)，当进位标志不是 1 时则跳转。但是，JCXZ 是一条喜欢孤独的指令。不存在 JCXNZ 指令，所以请不要试图到指令参考中去查找它！

11.5.4　关闭内层循环

假设 ECX 还没有被 STOSB 指令递减为 0 (一个 JCXZ 观察的条件)，循环仍会继续。AL 被递增。这就是一行中的下一个 ASCII 字符被选中的原理。AL 中的值被发送到由 STOSB 指令存放在 EDI 中的位置。如果递增 AL 中的值，那么将要显示的字符将变为一行中的下一个。例如，如果 AL 中包含 A 的值(65)，那么递增 AL 中的字符的值将使这个字符变为 B(66)。在下一轮通过这个循环时，STOSW 将一个 B 而不是 A 发射到屏幕上。

当 AL 中的字符码递增后，BL 被递减。现在，BL 并没有直接与字符串指令相关联。任何字符串指令制造的假设都不涉及 BL。这里，我们正在将 BL 用于某些其他方面。 BL 是作为一个计数器而存在的，它管理着屏幕上显示的字符行的长度。BL 已经早被加载了 CHRTLEN 等值所代表的值，它的值为 32。每经过一个循环，DEC BL 指令将 BL 中的值递减 1。然后 LOOPNZ 指令开始工作。

LOOPNZ 与前面讲述的 LOOP 指令有一点点不同。如果不真正了解它的工作原理，这点不同足以让你陷入麻烦。LOOP 和 LOOPNZ 都将 ECX 寄存器递减 1。LOOP 观察 ECX 寄存器的状态，直到 ECX 寄存器变为 0 时关闭循环。LOOPNZ 既观察 ECX 寄存器的状态，也观察零标志 ZF 的状态。(LOOP 忽略 ZF。)只有当 ECX < >(不等于)0 并且 ZF = 0 时，LOOPNZ 才会关闭循环。换句话说，只有当 ECX 中还剩下一些值，而且零标志 ZF 没有被设置时，LOOPNZ 才关闭循环。

那么，这里 LOOPNZ 指令究竟观察什么内容呢？记住，在 LOOPNZ 指令的紧前面，我们正通过一条 DEC BL 指令来将 BL 递减 1。DEC 指令经常影响 ZF。如果当某个 DEC 指令执行后，该 DEC 的操作数变为零，那么 ZF 则变为 1(被设

置)。否则，ZF 停留在 0(仍然保持清零状态)。所以，实际上，LOOPNZ 是在观察 BL 寄存器的状态。直到 BL 被递减为 0(设置 ZF)时，LOOPNZ 才关闭循环。当 BL 变为零后，内层循环完成， LOOPNZ 指令执行失败，程序执行流程跳转到下一个指令。

那么 ECX 呢？嗯，LOOPNZ 实际上在观察 ECX，但是 JCXZ 也是如此。JCXZ 实际上是一个开关，它管理着什么时候整个循环——不只内层部分，还有外层部分——已经完成其工作，并且必须停止。所以，尽管 LOOPNZ 实际上在观察 ECX，还有一些其他指令也在做那项工作，并且这些其他指令会在 LOOPNZ 采取行动之前对 ECX 采取行动。因此，LOOPNZ 的工作内容除了包括观察 BL 之外，还有递减 ECX。它管理着两个循环中的内层循环。

11.5.5　关闭外层循环

这是否意味着 JCXZ 关闭外层循环呢？不是的。JCXZ 指示什么时候这两个循环都完成。关闭外层循环采用了一些不同于关闭内层循环的办法。让我们再次看看这两个嵌套的循环：

```
.DoLn: mov bl,CHRTLEN ; 每一行将包含 32 个字符
.DoChr: stosb         ; 注意，没有 REP 前缀！
jcxz AllDone          ; 当整个序列打印出来后，退出
inc al                ; 将 AL 中的字符值增大 1
dec bl                ; 将行寄存器递减 1
loopnz .DoChr         ; 回去并处理另一个字符，直到 BL 变为 0
add edi,COLS-CHRTLEN  ; 将 EDI 移到下一行的起始位置
jmp .DoLn             ; 开始显示下一行
```

当我们将 ASCII 表中的一个完整的行(简称：ASCII 表行)显示到屏幕上时，内部循环被认为已经完成。BL 控制行的长度，当 BL 变为零时(LOOPNZ 指令检测它)，一行完成。LOOPNZ 指令执行失败，程序执行流程跳转到那条修改 EDI 的加法指令。

我们修改 EDI，让它从显示缓冲区中的一个已完成行的结尾跳到下一行开始的左边距处。这意味着我们必须要被一定数量的字符从 ASCII 表行的结尾“卷”到可视屏幕的末尾。这个所需的字节数由汇编时表达式 COLS-CHRTLEN 提供。这基本上是一个 ASCII 表行的长度和虚拟屏幕的宽度(而不是虚拟屏幕所展示在的终端窗口的宽度！)之间的差值。该表达式的结果就是为了到达下一行开始的屏幕左边距处而必须向显示缓冲区内部移动的字节数。

当通过修改 EDI 完成换行时，外层循环的工作已经完成，我们关闭循环。这一次，我们通过使用一条简单的 JMP 指令无条件地做这件事情。JMP 指令的目标是.DoL 局部标签。没有什么如果，也没有什么争论。在外层循环的顶部(由.DoLn 标签代表)，我们将一个表行的长度加载回现在的空空的 BL 中，然后恢复内层循环。内层循环再次开始向缓冲区中发射字符，并且持续这样做，直到 JCXZ 检测到 CX 已经变为 0。

就在那时，无论是内层还是外层循环都已经完成，完整的 ASCII 表已被写入了 VidBuff 中。做完这件事后，我们就可以通过调用 Show 过程把缓冲区发送到 Linux 控制台了。

11.5.6 Showchar 小结

让我们回顾一下刚才所做的事情，因为它被公认为相当复杂。showchar 程序包含两个嵌套的循环：内层循环通过 STOSB 向屏幕上发射字符。外层循环则通过重复执行内环一些次数(在这里为七次)，将字符行发往屏幕。

内层循环由 BL 寄存器中的值控制，这个值最初被设置为一行字符的长度(在这里为 32)。外层循环并没有显式地被即将显示的行数所控制。也就是说，我们不是将数字 7 加载到寄存器中，然后递减它。相反的，外层循环将一直继续，直到 ECX 中的值变成 0，表明整个工作——展示我们想要显示的所有 224 个字符——已经完成。

内层循环和外层循环都会修改 STOSB 工作的寄存器。每当一个字符被发射到屏幕上后，内层循环都修改 AL 中的值。这使得每次 STOSB 发射时可以显示一个不同的字符。每当 1 行字符完成显示后，外层循环都修改 EDI 寄存器(目标索引寄存器)的值。这使得我们能够将目标字符串分成 7 个独立的、非连续的行。

11.6 命令行参数和堆栈检查

当在 Linux 控制台的命令提示符下启动一个程序时，我们可以选择在可执行程序的路径名之后包含任意合理数量的参数。换句话说，我们可以这样执行一个名为 showargs1 的程序：

```
$./showargs1 time for tacos
```

这三个参数跟在程序的名称后面，并且用空格字符隔开。请注意，这些参数

都与 I/O 重定向参数不同，后者需要使用重定向操作符号，>或者<，并且被 Linux 单独处理。

当一个程序开始运行时，任何在程序加载时输入的命令行参数都被 Linux 传递给该程序。在本节中，我将解释一下 Linux 堆栈的结构，因为堆栈是命令行参数的存储位置。在下一节中，我们将会看到如何从一个汇编语言程序中访问一个程序的命令行参数。在这个过程中，我们将会看到另外一个 X86 字符串指令在行动：SCASB。

11.6.1 两块虚拟内存

早在第 8 章中我就从概念的层面上解释了 x86 堆栈。我一带而过了很多相关的细节，尤其是当执行一个程序时 Linux 设置堆栈的方式。现在是时候仔细看看了。

理解 Linux 堆栈至少需要对虚拟内存有大致的了解。Linux 操作系统经常使用虚拟内存机制来管理计算机中的物理内存。虚拟内存是少数需要详细解释的内容，从某一高度来看，它是这样工作的：Linux 可以拨出计算机物理内存系统中的任意一个区域，然后说，“你应该认为这块内存的第一个地址是 08048000h，并应据此对所有的内存进行相应的处理。”

这是一个谎言，但却是个有益的谎言。程序可以自由使用 Linux 分配给它的内存块，并假定它是唯一使用该内存块的程序。其他程序可能会被赋予自己的“虚拟”内存块，并且可以与你的程序同时运行。这些正在同步运行的程序没有一个意识到事实上也有其他程序正在运行，而且它们之间都互不干涉。

这是真正奇怪的部分：每个获得内存块的程序都可能被告知，它的内存块的地址从 08048000h 开始。即使对于同时运行的程序而言，也是这样。每个程序都认为，它运行在自己的内存小宇宙中，并且每个程序都自以为其内存地址开始于同一个地方。

这怎么可能呢？远在后台，Linux 内核接受任意程序代码发出的每一个单独的内存寻址尝试，并将那个虚拟地址转换成 RAM 中的某处物理内存地址。这涉及很多与物理内存表相关的工作，甚至(必要时)在硬盘驱动器上的存储位置“伪造”RAM 中的物理内存。但是底线是，程序将得到它自己的内存小宇宙，并且可以假设任意它拥有的内存真正为它自己所拥有。而那个内存在物理内存系统中的精确位置对于该程序而言，是不明晰的，而且也是不重要的。

当程序开始运行时，Linux 实施它的虚拟内存魔术，为该程序的代码和数据

预留内存区域。对于基于 x86 的 Linux 系统而言，这样的内存块总是始于 08048000h 处。从那里开始，它一路运行，一直到 0BFFFFFFFh(或其附近的某个地方；顶端地址并不总是相同，稍后会对此进行更多的介绍)。现在，那是一个很大的内存：超过 3GB。大多数个人电脑最近才达到 4GB 的已安装物理内存。那么，如果只安装了 2GB(例如)的内存的话，Linux 将如何交给每个程序 3GB 的内存来运行呢?

很简单：在这 3GB 的虚拟地址空间中，并不是每一个虚拟地址都映射到物理地址上。事实上，一个 Linux 程序的虚拟内存空间被划分成两块，如图 11-3 所示。较低的块始于 08048000h 并且包含程序的代码，以及在.data 段和.bss 段中定义的数据。它只有本身实际需要的那么大，由代码和已定义的数据来决定，对于本书展示的简单演示程序而言，它相当小——也许只有几百个字节。

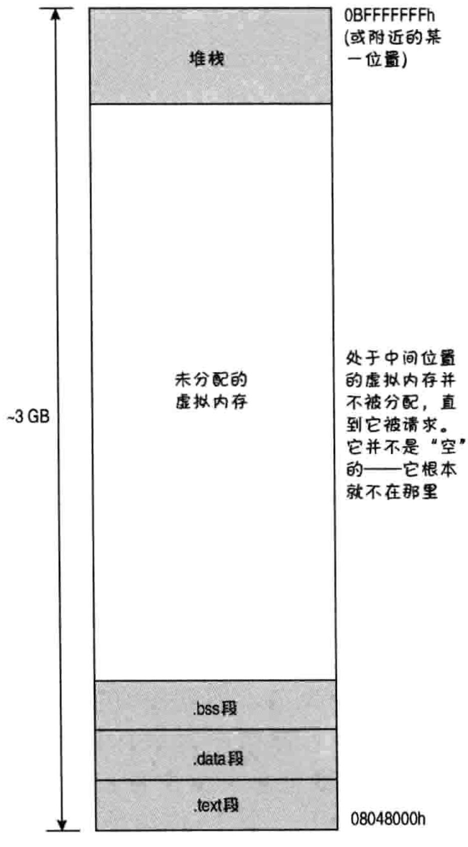


图 11-3　运行时程序的虚拟内存块

较高的块可被认为几乎正好相反：它从高内存开始，并且一路向低内存运行。这个较高的块的实际地址边界并不总是相同。但是，这个块的高端内存(有时被令人困惑地称为“栈底”) 不能高于 0BFFFFFFFh。这个较高的块就是程序的堆栈。

两个块儿之间的未使用区域表面上看起来很大，实际上是一种幻觉。如果程序需要从空空的中间区域中获取额外的内存，它只需试图在这一额外内存上寻址，Linux 将满足这个请求——除非该请求出于若干神秘的原因而被判定为太过分或者有缺陷。然后 Linux 就可以拒绝这个请求，并由于其行为越轨而终止这个程序。

一般来说，程序代码和数据将下降到某个位于(但是不低于)08048000h 附近的地方。堆栈将到达(但不高于)0BFFFFFFFh 附近的某个地方。

11.6.2 Linux 堆栈剖析

堆栈可能比你想象中的要大得多，也复杂得多。当 Linux 加载程序时，在开始执行该程序的代码之前，它将大量的信息放到堆栈里。这包括正在运行的可执行文件的完全合法的路径、用户执行程序时输入的任何命令行参数以及 Linux 环境的当前状态，它是一个文本配置字符串的集合，定义了 Linux 是如何配置的。

这些全部都有计划地列出来了，我已将其总结在图 11-4 中。首先让我们重温一些术语：堆栈的顶部(与直觉不符)是在图的底部。这是当程序开始运行时 ESP 所指向的内存位置。堆底是在图的顶部。它是当 Linux 加载并运行程序时赋给该程序的虚拟地址空间中的最高地址。有关“顶”和“底”的规定是一项古老的习俗，它经常会把很多人搞糊涂。内存原理图通常从页面底部的低内存开始，然后描绘它上面的更高的内存，虽然这意味着栈底是在原理图的顶部。一定要适应它，如果打算理解原理，你必须这样做。

Linux 从高内存向低内存生成堆栈，从堆栈的底部开始，去往下面的内存。当程序代码开始实际运行时，这里有一个在启动时堆栈内容的详细说明：

- 存放在 ESP 中的是一个 32 位的数字，用于指出存放在堆栈上的命令行参数的个数。此值始终至少为 1，即使没有参数输入。当执行程序时由用户输入的文本与所有命令行参数一起进行计数，而这个“调用文本”始终存在，这就是为什么计数值总是至少为 1 的原因。
- 位于 ESP 上面的内存中的下一个 32 位项是调用文本的地址，可执行文件通过它得到运行。该文本可能是完全路径，这意味着该文件的目录路径

名包括从/home 目录开始的路径，例如，/home/asmstuff/asm3ecode/showargs1/showargs1。这就是当我们从 Insight 中运行程序时，调用文本看起来的样子。如果使用“点斜杠”的方式从当前目录内调用可执行文件，我们将会看到可执行文件的名字前面有一个“./”前缀。

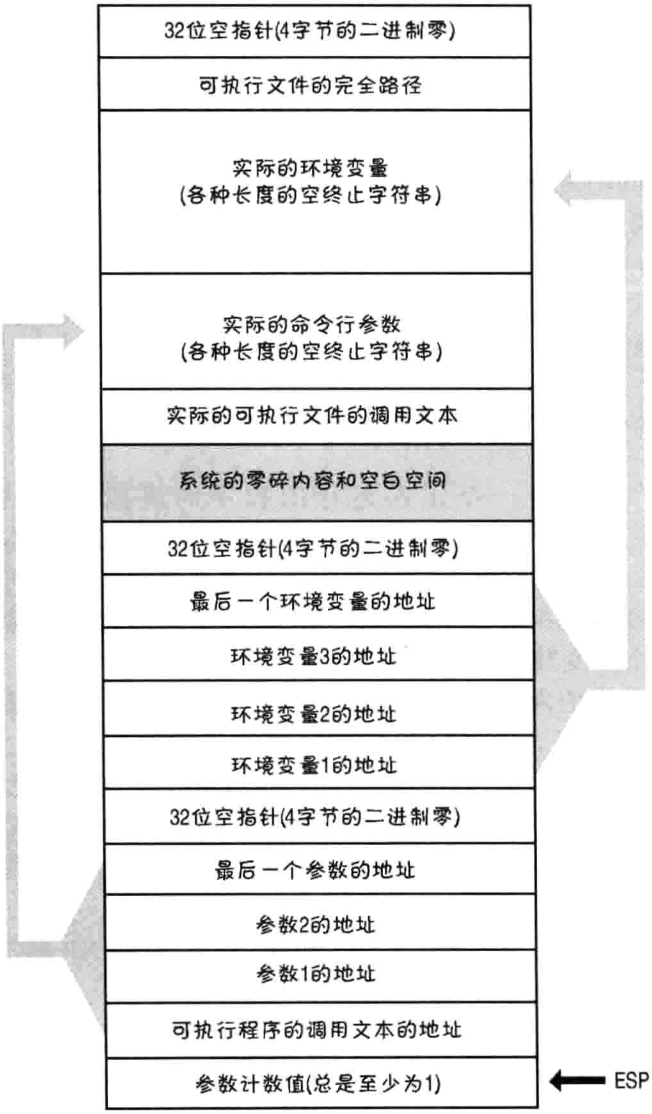


图 11-4 程序启动时的 Linux 堆栈

- 如果输入了任意数量的命令行参数，它们的 32 位地址都将位于从 ESP 开始的上面的内存中，第一个参数(最左边的)的地址后面跟着第二个参数的地址，等等。当然，参数的个数可能会有所不同，虽然我们很少需要超过 4 个或 5 个的参数。
- 该命令行参数的地址列表终止于一个空指针，这是 32 位的二进制 0 的术语。
- 从空指针上面的内存开始，是一个较长的 32 位地址列表。具体多少取决于特定的 Linux 系统，但是它可达近 200 个。这些地址每一个都指向一个以空(null)为终止字符的字符串(稍后会有更多关于它的介绍)，每一个里面都装载一个属于 Linux 环境的定义。
- 在 Linux 环境变量地址列表的末尾处，是另外一个 32 位的空指针，它标志着堆栈“目录”的结束。除此之外，你还可以使用之前在堆栈中找到的地址来访问上内存中更远一些位置上的数据项。

11.6.3 为什么堆栈的地址是不可预测的

几年以前，分配给一个正在运行的程序的虚拟空间的最高地址总是 0BFFFFFFFh，并且它总是堆栈的底部。从 Linux 内核的 2.6 版本开始，内核随机化了堆栈的界限。每次程序运行时，它的堆栈地址都会有所不同，通常相差数百万字节。此外，在地址列表的末尾和那些地址所指向的实际字符串数据的起始地址之间，还有一个可变数量的未使用内存“垫料”。

这使得阅读程序的内存布局更加困难，但是它出于一个很好的理由。如果堆栈总是位于同一个地址，缓冲区溢出可以强迫代码进入堆栈，使之作为数据存在，然后恶意软件就能够执行该代码。这种执行取决于该恶意代码的硬编码内存地址(因为没有加载器来调整代码地址)，但是如果堆栈(几乎)从来没有在同一个地方出现过两次，那么这个硬编码地址将不能工作，并且在堆栈上执行恶意代码会十分困难。

这就是 Linux 天生比那些较老的 Windows 版本(包括 XP 和 2000)更安全的诸多原因之一，后者的堆栈地址可以被很容易地猜到。

11.6.4 使用 Insight 设置命令行参数

一般来说，Gdb 的 Insight GUI 系统还是不错的，而且它的某些部分确实非常

优秀。但是，不幸的是，它的某些部分却纯粹令人感到不愉快。在很大程度上，这个不愉快来自内存视图窗口，它是使用 Insight 查看堆栈的唯一方法。

如果我们在运行 Insight 时附带了一个即将调试的程序的名称，这个程序并不会立即执行，而是存在于内存中，直到我们单击运行按钮。这使得我们有机会在程序运行之前设置断点。它也给了我们一个输入命令行参数的机会。要想在 Insight 下设置命令行参数，请打开控制台视图，在(gdb)提示符下的 set args 命令后面输入参数：

```
(gdb) set args time for tacos
```

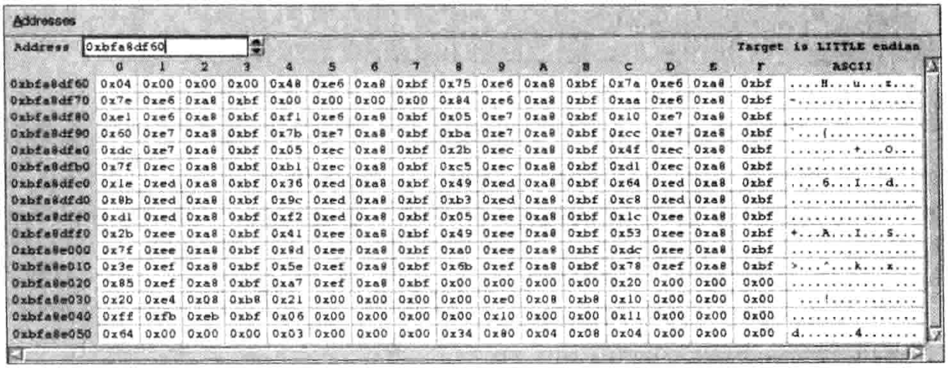
这里输入了三个参数：time、for 和 tacos。当在 gdb 控制台上输入了这些参数之后，按 Enter 键。要想验证输入的参数，可以通过 show args 命令从控制台视图查看它们。当所有必要的参数输入完毕后，我们可以单击 Run 按钮，让计算机暂停于第一个断点处。一旦程序开始运行，内存视图就活跃起来，并开始显示真实的数据。

11.6.5 通过 Insight 的内存视图查看堆栈

只有两种方法能够在存储器视图改变正在显示的内存区域的地址。一种是单击导航控制中的“向上”或“向下”箭头按钮。单击一次会将显示内容向上内存或者下内存移动 16 个字节。唉，这种方法的速度慢得令人痛苦。在一次单击一个箭头之后，它可能需要长达 3 秒的时间才能显示刷新后的结果。另一种方法是输入一个完整的 32 位地址(十六进制)到该导航控制地址输入栏字段。同样的，唉(也是令所有编写 Insight 的人感到惭愧的地方！)，该控件并不能接受从剪贴板粘贴的文本。

默认情况下，Insight 的内存视图被设置为从.data 段的第一个字节开始显示。要想查看堆栈，我们必须打开寄存器视图，然后将 ESP 中显示的地址输入到内存视图的导航控制。然后，内存视图将从顶部开始显示堆栈，也就是说，从 ESP 寄存器指向的字节处开始。

图 11-5 显示了 showargs1 程序的具有 3 个命令行参数的堆栈的存储器视图(参见下面的清单 11-3)。



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
0xbfa8df60	0x04	0x00	0x00	0x00	0x48	0xe6	0xa8	0xbf	0x75	0xe6	0xa8	0xbf	0x7a	0xe6	0xa8	0xbf	....H...u...z...
0xbfa8df70	0x7e	0xe6	0xa8	0xbf	0x00	0x00	0x00	0x00	0x84	0xe6	0xa8	0xbf	0xaa	0xe6	0xa8	0xbf	~...............
0xbfa8df80	0xe1	0xe6	0xa8	0xbf	0xf1	0xe6	0xa8	0xbf	0x05	0xe7	0xa8	0xbf	0x10	0xe7	0xa8	0xbf	....
0xbfa8df90	0x60	0xe7	0xa8	0xbf	0x7b	0xe7	0xa8	0xbf	0xba	0xe7	0xa8	0xbf	0xcc	0xe7	0xa8	0xbf	`...{...........
0xbfa8dfa0	0xdc	0xe7	0xa8	0xbf	0x05	0xec	0xa8	0xbf	0x2b	0xec	0xa8	0xbf	0x4f	0xec	0xa8	0xbf	........+...O...
0xbfa8dfb0	0x7f	0xec	0xa8	0xbf	0xb1	0xec	0xa8	0xbf	0xc5	0xec	0xa8	0xbf	0xd1	0xec	0xa8	0xbf	....
0xbfa8dfc0	0x1e	0xed	0xa8	0xbf	0x36	0xed	0xa8	0xbf	0x49	0xed	0xa8	0xbf	0x64	0xed	0xa8	0xbf	....6...I...d...
0xbfa8dfd0	0x8b	0xed	0xa8	0xbf	0x9c	0xed	0xa8	0xbf	0xb3	0xed	0xa8	0xbf	0xc8	0xed	0xa8	0xbf	....
0xbfa8dfe0	0xd1	0xed	0xa8	0xbf	0xf2	0xed	0xa8	0xbf	0x05	0xee	0xa8	0xbf	0x1c	0xee	0xa8	0xbf	....
0xbfa8dff0	0x2b	0xee	0xa8	0xbf	0x41	0xee	0xa8	0xbf	0x49	0xee	0xa8	0xbf	0x53	0xee	0xa8	0xbf	+...A...I...S...
0xbfa8e000	0x7f	0xee	0xa8	0xbf	0x8d	0xee	0xa8	0xbf	0xa0	0xee	0xa8	0xbf	0xdc	0xee	0xa8	0xbf	....
0xbfa8e010	0x3e	0xef	0xa8	0xbf	0x5e	0xef	0xa8	0xbf	0x6b	0xef	0xa8	0xbf	0x78	0xef	0xa8	0xbf	>...^...k...x...
0xbfa8e020	0x85	0xef	0xa8	0xbf	0xa7	0xef	0xa8	0xbf	0x00	0x00	0x00	0x00	0x20	0x00	0x00	0x00	....
0xbfa8e030	0x20	0xe4	0x08	0xb8	0x21	0x00	0x00	0x00	0x00	0xe0	0x08	0xb8	0x10	0x00	0x00	0x00	...!............
0xbfa8e040	0xff	0xfb	0xeb	0xbf	0x06	0x00	0x00	0x00	0x00	0x10	0x00	0x00	0x11	0x00	0x00	0x00	....
0xbfa8e050	0x64	0x00	0x00	0x00	0x03	0x00	0x00	0x00	0x34	0x80	0x04	0x08	0x04	0x00	0x00	0x00	d.......4.......

图 11-5 Insight 的存储器视图中的堆栈

在存储器视图中阅读堆栈显示的诀窍是要记住，数字和地址的大小都是 32 位，而且采用小端字节序(little – endian)显示。这意味着，字节是以它的含义的反向顺序显示的。换句话说，一个 32 位值的最低有效字节在前面，其次是比较重要的字节，等等。这就是为什么堆栈上的前四个值是如下内容的原因：

```
0x04 0x00 0x00 0x00
```

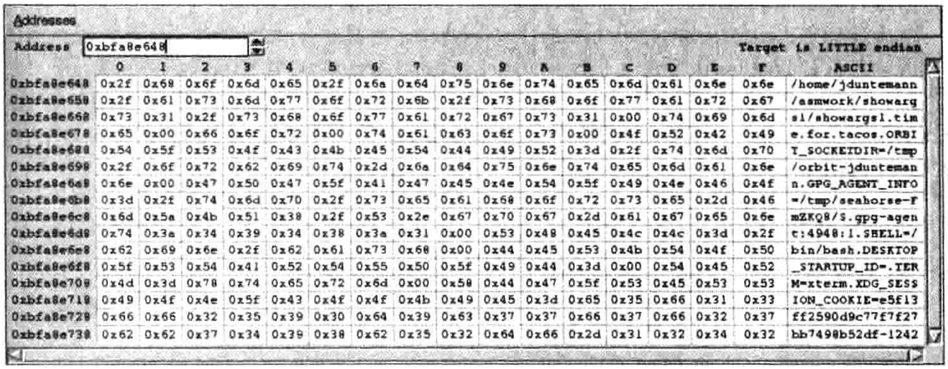
这是 32 位值 4，它是这三个命令行参数加上可执行文件的路径名的计数。

地址也是同样的。一个地址的最低有效字节第一个出来，所以 4 个字节的地址被按照曾经认为的 32 位地址“前后倒置”地使用。堆栈上的第一个地址，是调用文本的地址，执行文件就是通过它来运行。它看起来像这样：

```
0x48 0xe6 0xa8 0xbf
```

这 4 个字节代表 32 位地址 0BFA8E648h。将图 11-5 中的堆栈显示与图 11-4 中的堆栈原示意图关联起来。我们应该有计数值 4，其次是 4 个 32 位地址，随后又有四个 0 字节，它是 32 位的空指针。(这是一个源自 C 语言编程的术语，它有时被用于汇编语言工作，尽管我们一般情况下不将地址称为“指针”)在空指针的后面，我们将会看到一个额外的很长一段距离的地址，后面再跟着另一个空指针。

我们可以使用存储器视图来“跟踪”一个采用上内存方式存储的实际数据的地址。将堆栈上的第一个地址敲入导航控制中，然后视图将会移到该地址。在本例中，这应该是可执行文件的调用文本的地址(见图 11-6)。



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
0xbfa8e648	0x2f	0x68	0x6f	0x6d	0x65	0x2f	0x6a	0x64	0x75	0x6e	0x74	0x65	0x6d	0x61	0x6e	0x6e	/home/jduntemann
0xbfa8e658	0x2f	0x61	0x73	0x6d	0x77	0x6f	0x72	0x6b	0x2f	0x73	0x68	0x6f	0x77	0x61	0x72	0x67	/asmwork/showarg
0xbfa8e668	0x73	0x31	0x2f	0x73	0x68	0x6f	0x77	0x61	0x72	0x67	0x73	0x31	0x00	0x74	0x69	0x6d	s1/showargs1.tim
0xbfa8e678	0x65	0x00	0x66	0x6f	0x72	0x00	0x74	0x61	0x63	0x6f	0x73	0x00	0x4f	0x52	0x42	0x49	e.for.tacos.ORBI
0xbfa8e688	0x54	0x5f	0x53	0x4f	0x43	0x4b	0x45	0x54	0x44	0x49	0x52	0x3d	0x2f	0x74	0x6d	0x70	T_SOCKETDIR=/tmp
0xbfa8e698	0x2f	0x6f	0x72	0x62	0x69	0x74	0x2d	0x6a	0x64	0x75	0x6e	0x74	0x65	0x6d	0x61	0x6e	/orbit-jduntemann
0xbfa8e6a8	0x6e	0x00	0x47	0x50	0x47	0x5f	0x41	0x47	0x45	0x4e	0x54	0x5f	0x49	0x4e	0x46	0x4f	n.GPG_AGENT_INFO
0xbfa8e6b8	0x3d	0x2f	0x74	0x6d	0x70	0x2f	0x73	0x65	0x61	0x68	0x6f	0x72	0x73	0x65	0x2d	0x46	=/tmp/seahorse-F
0xbfa8e6c8	0x6d	0x5a	0x4b	0x51	0x38	0x2f	0x53	0x2e	0x67	0x70	0x67	0x2d	0x61	0x67	0x65	0x6e	mZKQ8/S.gpg-agen
0xbfa8e6d8	0x74	0x3a	0x34	0x39	0x34	0x38	0x3a	0x31	0x00	0x53	0x48	0x45	0x4c	0x4c	0x3d	0x2f	t:4948:1.SHELL=/
0xbfa8e6e8	0x62	0x69	0x6e	0x2f	0x62	0x61	0x73	0x68	0x00	0x44	0x45	0x53	0x4b	0x54	0x4f	0x50	bin/bash.DESKTOP
0xbfa8e6f8	0x5f	0x53	0x54	0x41	0x52	0x54	0x55	0x50	0x5f	0x49	0x44	0x3d	0x00	0x54	0x45	0x52	_STARTUP_ID=.TER
0xbfa8e708	0x4d	0x3d	0x78	0x74	0x65	0x72	0x6d	0x00	0x58	0x44	0x47	0x5f	0x53	0x45	0x53	0x53	M=xterm.XDG_SESS
0xbfa8e718	0x49	0x4f	0x4e	0x5f	0x43	0x4f	0x4f	0x4b	0x49	0x45	0x3d	0x65	0x35	0x66	0x31	0x33	ION_COOKIE=e5f13
0xbfa8e728	0x66	0x66	0x32	0x35	0x39	0x30	0x64	0x39	0x63	0x37	0x37	0x66	0x37	0x66	0x32	0x37	ff2590d9c77f7f27
0xbfa8e738	0x62	0x62	0x37	0x34	0x39	0x38	0x62	0x35	0x32	0x64	0x66	0x2d	0x31	0x32	0x34	0x32	bb7498b52df-1242

图 11-6　Insight 的存储器视图中的命令行参数

命令行参数和环境变量被首尾相连地存储在内存中。每一个都被一个单个的 0 字节(通常称为零)所终止，这也是为什么这样的字符串被叫作空终止字符串的原因。虽然没有技术上的理由，命令行参数和环境变量存储在堆栈的同一区域中，命令行参数在先，环境变量跟后。

我们可以在如图 11-6 所示的内存视图中的 ASCII 列里看到它们。紧跟在 tacos 参数的后面，我们会看到第一个环境变量：ORBIT_SOCKETDIR = tmp/orbit-jduntemann。它到底这意味着什么已经超出了这本书的范围(ORBIT 是一个 CORBA 对象请求代理)，但是 Linux 使用该变量来告诉那些需要知道某些基本的 ORBIT 文件存储在系统的什么位置的人。

11.7　使用 SCASB 进行字符串搜索

一旦了解了 Linux 堆栈在内存中如何布局，了解命令行参数就很容易了。我们拥有了位于堆栈上的相当于一张表的地址，每个地址指向一个参数。唯一棘手的内容是确定每个参数拥有多少个字节，以便于我们可以复制参数数据到任何其他需要的地方，或者把它传递给像 sys_write 那样的 Linux 内核调用。由于每个参数都以一个 0 字节结束，所以挑战很简单：我们要寻找 0。

这可以通过一种非常明显的方式来实现，在一个循环里从内存地址中读取一个字节，然后在将计数器值递增以读取下一个字节之前，将该字节与 0 进行比较。然而，好消息是，x86 指令集在一个字符串指令中就可以实现这样一个循环，该指令并不存储数据(如 STOSB)或复制数据，(如 MOVSB)，取而代之的

是，它能搜索内存以查找某一特定的数值。这条指令就是 SCASB (按字节扫描字符串)，如果已经掌握了其他字符串的相关资料，理解它应该是小菜一碟。

清单 11-3 通过查看堆栈上的命令行参数并生成一个参数长度表演示了 SCASB 的使用。然后，通过调用 sys_write，回送参数(与可执行文件的调用文本)给标准输出 stdout。

清单 11-3：showargs1.asm

```
; 可执行文件      : SHOWARGS1
; 版本            : 1.0
; 创建日期        : 4/17/2009
; 最后更新日期    : 5/19/2009
; 作者            : Jeff Duntemann
; 描述            : 一个用汇编语言编写的运行在 Linux 下的简单程序，
;                   使用 NASM 2.05,演示了访问堆栈中的命令行参数的方法。
;
; 使用这些命令生成：
; nasm -f elf -g -F stabs showargs1.asm
; ld -o showargs1 showargs1.o
;

SECTION .data             ; 包含已初始化数据的段
      ErrMsg db "Terminated with error.",10
      ERRLEN equ $-ErrMsg

SECTION .bss              ; 包含未初始化数据的段

; 本程序处理最多 MAXARGS 个命令行参数。
; 如果需要处理的参数个数比默认值 10 多，请修改 MAXARGS 的值，
; 从本质上来讲， 我们将参数的指针保存到一个基于 0 的数组中，
; 第一个参数指针是数组的第 0 个元素，
; 第二个参数指针是数组的第 1 个元素，等等。
; 访问参数及其长度的方式是这样的：
;       参数字符串：       [ ArgPtrs+ <index reg> * 4]
;       参数字符串的长度：[ArgLens + <index reg> * 4]
; 注意，当计算出参数长度之后，一个 EOL 字符(10h)被存放到每个字符串中的
;原本是空终止的地方。这使得通过 sys_write 打印出一个参数变得非常容易。
; 这不是必要的，如果喜欢在参数中保留 0 终止的话，你可以注释掉该行，
; 记住，如果在结尾处没有 EOL 字符，参数就不会正确地显示。

      MAXARGS equ 10         ; 我们最多支持#个参数
      ArgCount: resd 1       ; 传递给程序的参数的个数
```

```
        ArgPtrs: resd MAXARGS           ; 参数指针表
        ArgLens: resd MAXARGS           ; 参数长度表

SECTION .text                           ; 包含代码的段

global _start                           ; 连接器需要这个来找到入口点!

_start:
        nop                             ; 这个空操作指令让 gdb 感到很高兴

;从堆栈中获取命令行参数的个数并且验证它:
        pop ecx                         ; TOS 包含参数的个数
        cmp ecx,MAXARGS                 ; 判断一下参数数量是否超过 MAXARGS
        ja Error                        ; 如果这样,以一个出错信息而退出。
        mov dword [ArgCount],ecx        ; 将参数个数存入内存变量

; 一旦我们知道有多少个参数,循环将会把它们弹出到 ArgPtrs 中:
        xor edx,edx                     ; 将循环计数器归零
SaveArgs:
        pop dword [ArgPtrs + edx*4]     ; 弹出一个参数地址到内存表中
        inc edx                         ; 将计数器指向下一个参数地址
        cmp edx,ecx                     ; 计数器的值=参数的个数?
        jb SaveArgs                     ; 如果不等,回去继续执行下一轮循环

; 参数指针存放在 ArgPtrs 中,我们计算它们的长度:
        xor eax,eax                     ; 我们要查找 0,所以,将 AL 清零
        xor ebx,ebx                     ; 指针表的偏移地址从零开始
ScanOne:
        mov ecx,0000ffffh               ; 将查找限制在最大 65535 个字节以内
        mov edi,dword [ArgPtrs+ebx*4] ; 将被搜索的字符串的地址放入 EDI 中
        mov edx,edi                     ; 将起始地址复制到 EDX 中
        cld                             ; 将查找方向设置为上内存
        repne scasb                     ; 在存放在 edi 中的字符串里查找
                                          null(0 字符)
        jnz Error                     ; REPNE SCASB 以没有找到 AL 而结束
        mov byte [edi-1],10           ; 存放一个 EOL 到 null 曾经存在的地方
        sub edi,edx                   ; 将 0 的位置从开始地址中减除
        mov dword [ArgLens+ebx*4],edi     ; 将参数的长度放入表中
        inc ebx                       ; 将参数计数器加 1
        cmp ebx,[ArgCount]            ; 判断参数计数器是否超过参数总数
        jb ScanOne                    ; 如果没有超过,回去继续执行另一轮循环
```

```
; 将所有的参数显示到 stdout:
        xor esi,esi                  ; 从 0 开始(出于表寻址的原因)
Showem:
        mov ecx,[ArgPtrs+esi*4]      ; 传递消息的偏移地址
        mov eax,4                    ; 指定 sys_write 调用
        mov ebx,1                    ; 指定文件描述符: 标准输出
        mov edx,[ArgLens+esi*4]      ; 传递消息的长度
        int 80H                      ; 进行内核调用
        inc esi                      ; 递增参数计数器
        cmp esi,[ArgCount]           ; 判断一下我们是否已经显示了所有的参数
        jb Showem                    ; 如果没有, 回去继续执行下一轮循环
        jmp Exit                     ; 大功告成! 让我们停止它!
Error:  mov eax,4                    ; 指定 sys_write 调用
        mov ebx,1                    ; 指定文件描述符 2: 标准错误
        mov ecx,ErrMsg               ; 传递错误信息的偏移地址
        mov edx,ERRLEN               ; 传递消息的长度
        int 80H                      ; 进行内核调用
Exit:   mov eax,1                    ; 退出系统调用的代码
        mov ebx,0                    ; 返回一个零代码
        int 80H                      ; 进行内核调用
```

showargs1 程序首先从堆栈中弹出参数数量到 ECX 中，如果不超过 MAXARGS 中的值，那么 ECX 中的计数值就被用于管理一个循环，该循环将参数本身的地址弹出到双字表 ArgPtrs 中。我们以后可以使用此表来访问参数本身。

REPNE SCASB 指令用来查找每个参数结尾的 0 字节。设置 SCASB 与设置 STOSB 大致相同：

- 对于“上内存”搜索(像这个示例)，使用 CLD 指令确保方向标志 DF 被清除。
- 将要进行搜索的字符串的第一个字节的地址放入 EDI 中。在这里，它是一个堆栈上的命令行参数的地址。
- 将即将被查找的值放入 AL 中(这里，它是 0)。
- 将最大计数值放入 ECX 中。这样，如果我们查找的字节实际上不在那里的话，就可以避免在内存中搜索过远的范围了。

所有内容都各就各位之后，REPNE SCASB 就能够执行了。如同 STOSB 一样，它将创建一个 CPU 内部的紧密循环。在每一轮循环中，[EDI]处的字节都与 AL 中的值进行比较。如果相等，则循环满意(地结束)，REPNE SCASB 停止执

行。如果不相等，则 EDI 被递增 1，ECX 被递减 1，继续循环，测试另一个位于[EDI]处的字节。

当 REPNE SCASB 找到 AL 中的字符并结束后，EDI 将指向字符串中那个被搜到的字符后面位置的字节。要想访问找到的字符，必须从 EDI 中减去 1，就像该程序将 0 字符替换为一个 EOL 字符时所做的那样：

```
mov byte [edi-1],10        ; 在 null 曾经存在的位置存储一个 EOL
```

11.7.1 REPNE v. s. REPE

SCASB 指令与 STOSB 和 MOVSB 略有不同，它是有条件的字符串指令。当前面带有 REP 前缀时，STOSB 和 MOVSB 都无条件地重复。除了测试 ECX 以判断循环是否已经进行了预定义次数的迭代之外，不进行其他的测试。与此相反，每次“射击”时，SCASB 都执行一个单独的测试，并且每个测试都可以以两种方式进行。这就是为什么我们不是对 SCASB 使用无条件前缀 REP，而是使用 REPNE 前缀或 REPE 前缀的原因。

当我们正在寻找搜索一个字符串以查找与 AL 中的值相匹配的字节时，使用 REPNE 前缀，在 showargs1 中就是这样做的。当我们正在搜索一个字符串以查找与 AL 中的值不相匹配的字节时，使用 REPE 前缀。你可能会认为这听起来有点正好相反，的确是那样的。然而，对 REPNE 前缀的理解是这样的：只要[EDI]不等于 AL，就重复执行 SCASB。同样，对 REPE 前缀的理解是这样的：只要[EDI]等于 AL，就重复执行 SCASB。该前缀表明 SCASB 指令应继续“射击”多久，而不是应该何时停止。

REPNE SCASB 可以由于两个原因之一而结束，记住这一点非常重要：它找到了与 AL 中的值相匹配的字节或者计数 ECX 已经减少到 0。在几乎所有的情况下，如果 ECX 为零时 REPNE SCASB 结束，都意味着没有在搜索字符串中发现 AL 中的字节。但是，有一种侥幸的可能性，即 ECX 刚好递减到零时，[EDI]中包含与 AL 中的值相匹配的字节。这似乎不太可能，但也存在一些发生这种情况的数据组合。

每次 SCASB“射击”时，它都做一次比较，该比较或者设置零标志 ZF 或者将其清零。当 REPNE 指令的比较将 ZF 设置为 1 时，该指令将会结束。当 REPE 指令的比较将 ZF 设置为 0 时，该指令将会结束。但是，要想确保绝对捕获搜索失败的结果，我们必须在 SCASB 指令结束后立即测试 ZF。

对于 REPNE SCASB：使用 JNZ。

对于 REPE SCASB：使用 JZ。

11.7.2 从堆栈中弹出，还是对堆栈寻址

showargs1 程序演示了一种比较明显的访问堆栈内容的方式：将它们从堆栈中弹出到寄存器中或者变量中。还有另外一种方式，这是提出一种有趣的程序设计问题的好地方：是将数据从堆栈中弹出到寄存器或者变量中这种方式好一些，还是通过内存引用来访问存储在堆栈中的数据，将数据本身留在原地好一些呢？记住，一旦数据位于堆栈之中，它并不被以任何特殊方式对待。堆栈数据是在内存中，并且像程序内存空间中的任意一个位置一样，可以使用任意一种合法的寻址方式在括号内寻址。

另一方面，一旦从堆栈中弹出一个数据项到寄存器中，从技术上来讲，它就已经不再位于堆栈之上了，因为 ESP 已经向上内存的方向移动了数据项大小那么远。诚然，堆栈中的数据不会因为从堆栈中弹出而被覆盖，但是当下一次任何数据被压入堆栈时，ESP 中的地址的下内存数据将被覆盖。如果这是弹出的数据项所在的地方，那么该数据项将会消失，并且被新的堆栈数据所取代。

要想对堆栈上的数据进行寻址，唯一棘手的地方是：栈顶在程序的运行过程中，并不总是位于同一个地方。如果程序调用了过程，那么 ESP 中的值将会随着返回地址被压入堆栈然后再从堆栈中弹出而发生改变。代码也能够将一些值压入堆栈用于临时存储。依靠 ESP 的一个不变的值来对堆栈进行寻址，这是非常冒险的，除非在非常简单的程序中。

在大多数应用中，一个更好的方法是：只要程序一开始运行，就将 ESP 复制到一个 32 位的指针寄存器中，创建一个不变的堆栈指针的副本。完成这件事情的一个不错选择是 EBP，也就是 16 位 BP(基指针)寄存器的 32 位版本。BP 最初被设计用来做的事情就是保存堆栈的指针，从而使以后“非破坏性的”堆栈数据寻址不会因为堆栈操作而变得更加困难。如果在做任何能够改变 ESP 值的事情之前，将 ESP 复制到 EBP 中，我们就拥有了一个堆栈“书签”，因而可以根据它来对堆栈深处的任意数据进行寻址。

我创建了 showargs1 程序的一个稍微不同的版本以演示这一工作原理。全部的程序——showargs2 存放在本书的存档列表中，但是相关的代码如下所示。将它与 showargs1 比较一下，看看两个程序有什么不同：

```
_start:
        nop                             ; 这个无操作指令让 gdb 感到非常高兴

        mov ebp,esp                     ; 将初始堆栈指针保存到 EBP 中
; 验证命令行参数数量:
        cmp dword [ebp],MAXARGS         ; 判断一下参数个数是否超出了 MAXARGS
        ja Error                        ; 如果这样，以一个错误信息而退出

; 这里，我们计算参数的长度并将其存储到表 ArgLens 中:
        xor eax,eax                     ; 我们要查找 0，所以将 AL 清零
        xor ebx,ebx                     ; 堆栈地址偏移从 0 开始
ScanOne:
        mov ecx,0000ffffh               ;将查找限制在最多 65535 个字节之内
        mov edi,dword [ebp+4+ebx*4]     ; 将要被搜索的字符串的地址放入 EDI 中
        mov edx,edi                     ; 将起始地址复制到 EDX 中
        cld                             ; 将查找方向设置为上内存(up-memory)
        repne scasb             ; 在 edi 中的字符串中查找 null (0 字符)
        jnz Error               ; REPNE SCASB 以没有找到 AL 中的值而结束
        mov byte [edi-1],10     ; 在 null 曾经存在的地方存入一个 EOL 字符
        sub edi,edx             ; 从开始地址减去 0 的位置
        mov dword [ArgLens+ebx*4],edi     ; 将参数的长度放入表中
        inc ebx                           ; 将参数计数器加 1
        cmp ebx,[ebp]           ; 判断一下参数计数器是否超过了参数个数
        jb ScanOne              ; 如果没有，回去继续循环，进行另一轮处理

; 将所有的参数显示到标准输出:
        xor esi,esi                 ; 从零开始(出于表寻址的原因)
Showem:
        mov ecx,[ebp+4+esi*4]       ; 传递消息的偏移量
        mov eax,4                   ; 指定 sys_write 调用
        mov ebx,1                   ; 指定文件描述符 1: 标准输出
        mov edx,[ArgLens+esi*4]     ; 传递消息的长度
        int 80H                     ; 进行内核调用
        inc esi                     ; 递增参数计数器
        cmp esi,[ebp]               ; 判断一下是否已经显示了所有参数
        jb Showem                   ; 如果没有，回去继续循环，进行另一轮处理
        jmp Exit                    ; 大功告成！让我们停止它！
```

在程序的最开始部分，我们通过将 ESP 复制到 EBP 创建了一个堆栈指针的不变副本。所有对于堆栈的其他访问都经由 EBP 实现。变量 ArgPtrs 和 ArgCount 都不见了。在程序 showargs1 中，存放在这两个变量之中的数据可以通过堆栈来

获得，在 showargs2 中，我们将其留在堆栈上，并从堆栈中使用它。

在程序启动时，栈顶包含参数计数，因此访问这个参数计数毫无问题：它是存放在[EBP] 中的双字量。为了访问堆栈中更远处的数据项，(如参数地址)，我们必须将偏移量与 EBP 相加。参数计数上面的 4 个字节是第一个参数的地址，所以这个参数的有效地址是[EBP+4]。每在堆栈中向更远处上移 4 个字节是另一个参数的地址，直到我们遇到了空指针。如果我们将 EBX 寄存器用作一个基于 0 的参数计数器，那么任意给定参数的地址可以这样计算：

```
[EBP +4 + ebx * 4]
```

这就是我们使用基于堆栈的参数地址来找到参数的长度，然后再将它们显示到控制台上的原理。不需要任何弹出操作，并且，任何在程序开始运行时存放在堆栈上的数据仍然存放在那里，以防我们在后续程序中需要再次使用它们。

11.7.3　额外的学分

实践出真知，所以这里有一个挑战，你可以在自己的程序中实现它：重写 showargs2.asm，让它能够显示出 Linux 环境变量的完整列表，而不只是显示程序的命令行参数。大部分的挑战在于如何对堆栈进行寻址。环境变量的地址位于 EBP 上面的内存中，但是其距离 EBP 的偏移量却不是固定不变的。它们位于什么地方取决于用户输入了多少个命令行参数。

它实际上比听起来要更容易一些。即使是一个线索也能够将其“出卖”，所以一定要动动脑筋，看看你能想出什么。(你可以在本书在线存档清单中找到答案，但是一定要先自己试一试！)

第 12 章

C 语言

调用 C 语言编写的外部函数

学习汇编语言有很多方面的价值，其中大部分源自这样的需求：我们必须详细知道一切是如何工作的，否则无法深入学习相关知识。这一直是正确的，甚至从最早的曙光数字电子计算机开始，但是它也引出了一个合理的问题：难道我真的需要知道所有那些内容吗？

平心而论，答案是否定的。即使对计算机和操作系统没有汇编级别的掌握，我们照样也可以编写出非常有效的程序。这就是为什么高级语言被创建成允许更容易、更快速地在一个更高的抽象水平上编程的原因。如果当今存在的软件都必须完全用汇编语言编写，那么它们到底该有多大恐怕很难搞清。

这些软件中包括 Linux 操作系统。Linux 的一小部分是用汇编语言编写的，但是就总体而言，大部分操作系统是用 C 语言编写的。Linux 宇宙围绕着 C 语言而“转动”，如果期望能够在 Linux 下很好地使用汇编语言，最好准备学习 C 语言，并且必要时使用它。

有一个几乎立即就能获得的回报：能够访问 C 语言编写的函数库。有成千上万个此类库文件，并且那些与 Linux 操作系统相关的大多是免费的，而且提供 C 源代码。使用 C 函数(译者注：汇编语言中的过程在 C 中被称为函数)库有优点也有缺点；而学习 C 函数调用相关技术的真正原因在于它是知道所有内容如何工作的一部分，尤其是在 Linux 操作系统下，几乎你能看到的所有地方都留下了 C 语言的痕迹。

你将发现，几乎所有针对 Linux 的、不涉及诸如 Perl 或 Python 之类的解释语言的编程示例都是用 C 编写的。最重要的是，C 运行时库包含了很多非常有用的函数，但是在调用这些函数时我们需要使用 C 协议。所以，如果你对语言还不了解，先买本书，坐下来猛学一通 C 知识。你不必为之做很多，但是要确保了解

C 的所有基本概念，特别是应用于函数调用的那些。在本书中，我会尽力填补低级差距，但是我不能教给你该语言本身，也不能教给你所有随之而来的各种约定。你可能会发现 C 有点令人烦(像我一样)，或者你也许会喜欢它，但是你无法逃避它，即使你对 Linux 的主要兴趣是汇编语言。

12.1 什么是 GNU

早在 70 年代末期，一个名叫理查德·斯托尔曼的狂妄的 Unix 黑客希望拥有自己的 Unix 副本。但是，他不想花钱，所以他做了这件显而易见的事情：开始编写他自己的版本。(如果它对你而言并非显而易见，那么，你还不明白 Unix 的文化。) 不过，他对所有现有的编程工具都不满意，并且对其较高的成本也非常反对。因此，作为编写自己的 Unix 版本的一个先决条件，斯托尔曼着手编写自己的编译器、汇编编译器和调试器。(他也已经编写了自己的编辑器——传说中的 Emacs。)

斯托尔曼将他的 Unix 版本命名为 GNU，一个递归缩写，意思是 GNU 并不是 Unix。这是一个很好的讽刺，也是一种能够让 AT&T 商标律师不再找茬的很好方式，那帮人在那些日子里一直在挑剔谁使用了 Unix 这个词以及如何使用它。随着时间的推移，GNU 工具不断发展壮大；并且因为这件事情的发生，斯托尔曼从来也没有真正完成编写 GNU 本身。其他的 Unix 免费版本出现了，并且几年之后，就谁真正拥有这些版本的哪些部分这一主题还发生了一些肥皂剧。这使得斯托尔曼非常反感，于是他创立了自由软件基金会作为 GNU 工具开发的大本营，还有一个激进的软件许可，名为 GNU 公共许可(GPL)，有时被非正式地称为“开放版权”。斯托尔曼在 GPL 下发布了 GNU 工具，这不仅需要该软件是免费的(包括所有的源代码)，而且阻止人们因对该软件作了轻微的修改便将其衍生作品称为自己所有。修改和改进都必须归还给 GNU 社区。

在当时看来，这似乎已经足够潇洒了，但是在出现之后的几年里，它便承担了自己特殊的逻辑和使命。GPL 已经允许发布在它下面的软件非常迅速地进化，因为许多人都使用它并对其加以改善，而且将这些改进免费或无限制地给回。于是，在这口沸腾的开源大锅中终于出现了 Linux，最初的 GPL 操作系统。Linux 采用 GNU 工具集构建和维护。如果打算在 Linux 下编程，无论使用什么语言，你迟早会用到一个或更多个 GNU 工具。

12.1.1　“瑞士军刀”编译器

在现代的 Linux 发行中，你会发现，Emacs 副本并不完全是理查德 • 斯托尔曼遗留在它里面的——它已经在过去的 20 多年中被很多其他人重写了几百遍。斯托尔曼的“遗产”主要包含在 GNU 编译器系列中。这类“遗产”有很多，但是，你必须竭尽全力去详细了解的一个可能是 GNU 的 C 编译器：gcc。(Unix 世界有点盲目崇拜小写字母，很多人都不理解为什么会这样，包括我在内。)

在使用汇编语言工作时为什么要用到 C 编译器呢？主要是这样：gcc 所做的事情不仅限于简单地编译 C 代码。它是一种瑞士军刀式的开发工具。事实上，将它所做的事情描述为生成软件而不仅仅是编译软件，会更恰当些。除了将 C 代码编译为目标代码之外，gcc 既管理编译步骤，又管理连接步骤。

编译步骤？是的，千真万确。就有这么一款 GNU 汇编编译器，名为 gas，尽管它并不打算真的供人类程序员所用，这一点有点奇怪。gcc 的工作就是控制 gas 和 GNU 连接器 ld(你已经使用过了)，就像 puppet 对字符串的管理一样。如果你在使用 gcc，尤其是在初级水平上使用它，不必直接围绕 gas 和 ld 做太多的事情。

让我们来更多地谈谈这个主题。

12.1.2　以 GNU 的方式生成代码

汇编语言的工作与 C 的工作相差很大，gcc 是第一个也是最重要的一个 C 编译器。因此，我们需要首先看一下生成 C 代码的过程。从表面上看，使用 GNU 工具为 Linux 生成一个 C 程序非常简单。但是，在幕后，它却是一项非常棘手的业务。虽然看起来像是 gcc 在做所有的工作，但实际上 gcc 真正做的事情是作为几种 GNU 工具的主控制器，监督除了你特别想看的之外的、根本不需观察的代码装配线。

从理论上讲，要从 C 源代码生成一个可执行的二进制文件，需要做的所有事情如下：

```
gcc eatc.c -o eatc
```

在这里，gcc 拿到 eatc.c 文件(这是一个 C 源代码文件)，并“压碎”它以产生可执行文件 eatc。(-o 选项告诉 gcc 如何为可执行文件命名。)但是，这里还有

比我们能够看到的多得多的事情发生。当我们通过它的时候，请看一下图 12-1。在图中，带阴影的箭头表示信息的流动。空白箭头表示程序的控制。

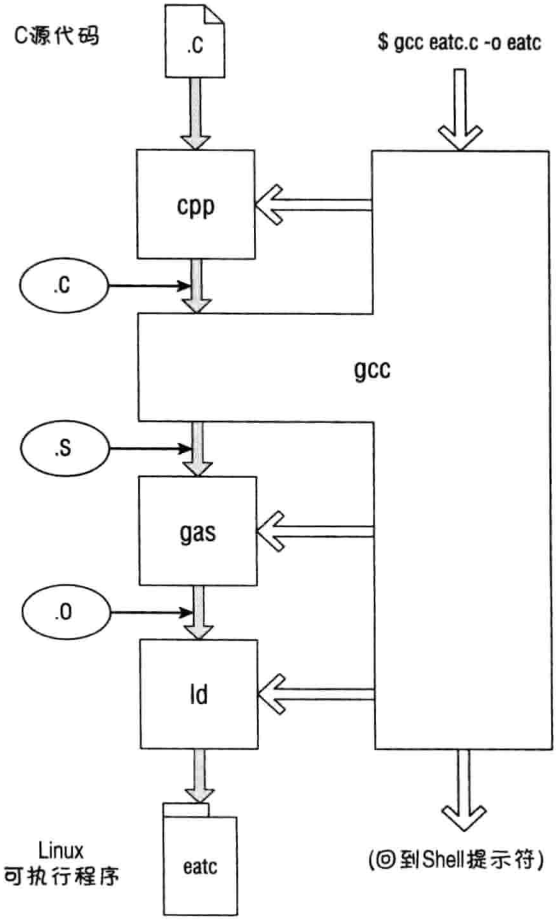


图 12-1　gcc 生成 Linux 可执行文件

程序员通常在终端窗口中通过 shell 命令行调用 gcc。gcc 接管了系统控制之后，立即调用一个名为 C 预处理器的实用工具：cpp。预处理器获得最初的 C 源代码文件，并处理某些诸如 # includes 和 # defines 之类的项。它可被认为是一种传递给 C 源代码文件的宏扩展。

每当 cpp 完成它的工作后，gcc 会认真地接管过来。gcc 从预处理 C 源代码文件生成一个以.s 为扩展名的汇编语言源代码文件。以人类可读的形式来看，它

几乎是与原来的.c 文件中的 C 语句相等的汇编代码。如果想培养任何阅读 AT&T 汇编语法和助记符(后面会有更多关于这一点的讲述)的技巧，研究 gcc 生成的.s 文件就能够学到很多。

当 gcc 已完全生成了与 C 源代码文件相等的汇编语言文件后，它调用 GNU 汇编编译器，gas，将.s 文件编译为目标代码。然后这个目标代码被写出到一个以.o 为扩展名的文件中。

最后一步涉及连接器：ld。.o 文件包含二进制代码，但是，这只是从最初的.c 文件生成的二进制代码。该.o 文件不包含来自在 C 编程中非常重要的标准 C 函数库代码。这些库已经被编译好，只有在需要时才连接到应用程序中。连接器 ld 在 gcc 的指导下完成这项工作。好的部分是：gcc 能够清楚地知道应用程序需要连接到哪些标准的 C 函数库才能工作，并且它始终包括正确版本中的正确的库。因此，虽然 gcc 并没有做实际的连接工作，但是它知道需要连接什么——随着程序变得越来越复杂，这一点确实非常有价值。

在行的结尾处，ld 给出了充分连接的、可执行的程序文件。此刻，生成工作已经完成，gcc 将控制返回给 Linux shell。注意，所有这一切通常使用一个简单的 gcc 命令就可以实现！

12.1.3 如何在汇编工作中使用 gcc

我刚才所描述的过程以及以图形方式在图 12-1 中演示的过程是如何在 Linux 下使用 GNU 工具生成一个 C 程序。这里，我们将走进一些细节，因为我们将要使用其中一部分，——虽然只是部分，却可以使我们的汇编编程变得更加容易。我们不需将 C 源代码转换为汇编代码，这是对的——事实上，我们不需要 gas 将 gas 汇编源代码转换为目标代码。不过，我们确实需要 gcc 在连接方面的专长。我们将深入了解连接阶段的 GNU 代码生成过程，以便 gcc 能够为我们协调连接步骤。当使用 NASM 编译一个 Linux 程序时，NASM 生成一个包含二进制目标代码的.o 文件。在 Linux 下调用 NASM 可以采用这样的典型方式实现：

```
nasm -f elf -g -F stabs eatclib.asm
```

此命令指示 NASM 编译 eatclib.asm 文件，并生成一个名为 eatclib.o 的文件。-f elf 部分告诉 NASM 生成 ELF 格式的目标代码，而不是 NASM 能够产生的众多其他目标代码格式之一。-g -F stabs 部分使得输出文件能够以 STABS 格式产生调试信息。eatclib.o 文件本身并不是可执行文件。它需要被连接。因此，我

们调用 gcc 并且指示它为我们连接程序：

```
gcc eatclib.o -o eatclib
```

到底是哪个部分告诉 gcc 进行连接，而不是编译呢？在该命令中显示的唯一输入文件是一个名为.o 的包含目标代码的文件。这一事实本身告诉 gcc 所有需要做的事情就是将该文件与 C 运行时库相连接，产生最后的可执行程序。- o eatclib 部分告诉 gcc，最终的可执行文件的名称为“eatclib”。

包含-o 标识符这一点非常重要。如果不精确地告诉 gcc 最后的可执行文件的名称，它会即兴创作并将其命名为“a.out”。

12.1.4 为什么不用 gas

你可能想知道，如果已经有了一个能够随着每一个 Linux 拷贝自动安装的、非常完美的汇编编译器，为什么我还要不辞辛苦地向你展示如何安装和使用另外一个呢？这是出于以下两个原因：

- GNU 汇编编译器 gas 使用一种特殊的语法，它与所有在 X86 世界里使用的其他我们熟悉的汇编编译器完全不同，包括 NASM。它有一整套自己独一无二的指令助记符。我觉得它们既丑陋，又不直观，而且难以阅读。这就是 AT&T 的语法，之所以这样命名，是因为它是由 AT&T 公司创建的一种可移植式汇编表示法，目的是使 Unix 系统能够更加容易地由一种底层 CPU 移植到另外一种上。它之所以丑陋，部分原因是它被设计为通用的，它可以为任何可能出现的、合理的 CPU 架构而改写。
- 更重要的是，“可移植式汇编”表示法在我看来是自相矛盾的说法。汇编语言应该是对于底层机器架构的直接的、彻底的、一对一的映射。任何使汇编语言通用化的企图都会使得该语言偏离机器，并且限制了汇编程序员像 CPU 原本被设计的那样指挥 CPU 的能力。创造和发展 CPU 架构的组织应该是定义 CPU 助记符的当之无愧的最佳人选。这就是为什么我会一直使用和讲授英特尔助记符的原因。

如果只是这么简单，我根本不会提及 gas，因为在 NASM 下编写 Linux 汇编语言根本不需要 gas。但是，我们最终学习标准 C 库文件调用的一个主要方法就是通过在简短的 C 程序中使用它们，然后检查 gcc 产生的.s 汇编输出文件。当你正处于熟悉 C 调用约定期间，具有一些阅读 AT&T 助记符的能力是很有用的。我将稍后在本章中提供一个 AT&T 语法概要。

12.2 连接到标准的 C 函数库

当编写一个(像那些到目前为止我在本书中介绍的示例一样的)全汇编程序时，所有的一切都需要我们自己编写。除了偶尔潜水到 Linux 内核服务之外，所有运行的代码都是我们自己编写的代码。连接到外部汇编语言过程会使情况变得有点复杂，特别是当你不是那个编写库的人时。连接到标准 C 函数库(这在 Linux 下被叫作 glibc) 则会使情况变得复杂得多。

事实上，当你创建一个 Linux 汇编语言程序，并将其与 glibc 中的函数连接在一起时，你创造了一种类型的“混血儿”。这个混血儿的结构如图 12-2 所示。

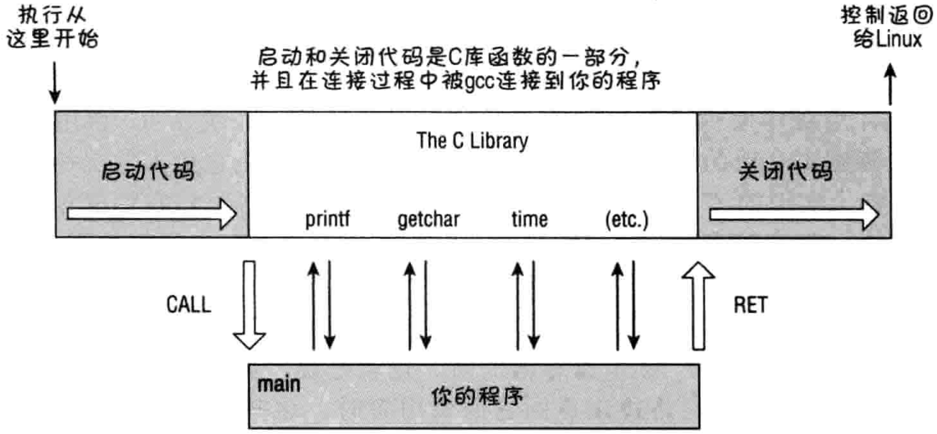


图 12-2 一个 C-汇编“混血”程序结构

你的程序已经不再像以前的汇编程序那样，是一件简单的、自顶往下的事情。glibc 不只是一个互相独立的函数的集合。它是标准的 C 运行时库，并且作为标准的一部分，它决定了任何一个连接到它的程序的结构。该结构包含一个在程序开始之前运行的代码块，和一个在程序结束之后运行的代码块。启动代码像调用过程一样(通过 CALL 指令)调用你的程序，而你的程序则通过使用 RET 指令将控制返回给 C 代码库。

从技术上来讲，程序是一个过程，并且把它看成一个过程也会有助于对它的理解。这就是图 12-2 所依据的原理。当 Linux 开始执行你的程序时，它实际上并不是从你编写的代码的顶部开始，而是从启动代码块的开头开始。当启动代码已经做完了它必须做的事情之后，它将执行一条 CALL 指令，把执行带到汇编代

码。当汇编语言程序通过 RET 指令将控制返回给调用它的程序时，关闭代码开始执行，实际上，正是关闭代码通过必要的 INT 80h 内核调用，将控制权返回给 Linux。

在此期间，想做多少对 glibc 的调用都可以。当使用 gcc 连接你的程序时，你所调用的包含 C 库例程的代码被连接到程序中。一定要注意，启动、关闭代码以及所有用于库调用的代码实际上存在于 gcc 生成的可执行文件中。

12.2.1　C 调用公约

glibc 并不是唯独挑出汇编语言程序来特殊对待。纯粹的 C 程序也以相同的方式工作，这就是为什么 C 程序的主程序部分被称为主函数的原因。它的确是一个函数，标准的 C 库代码调用通过一条 CALL 指令来调用它，然后它通过执行 RET 指令将控制返回给标准 C 代码库。

因此，主程序取得控制权的方式是你将看到的第一个关于 C 调用规则的例子。如果规则不一致的话，标准 C 库什么都不是，这也是它最大的美德。所有在 x86 处理器上实现的 C 库函数都遵循这些规则。一定要尽早将它们深刻地记在脑子里，这样，你在努力理解它们时所花费的时间就会比我当年用的少很多了。

下面是从某一高度对这些规则的总结：

- 过程(这是 C 调用函数的更为通用的术语)必须保存 EBX、ESP、EBP、ESI 和 EDI 这些 32 位寄存器的值。也就是说，虽然它可能会使用这些寄存器，但是当它将控制返回它的调用者时，这些寄存器的值都必须与它们在该函数调用之前的值相同。其他所有通用寄存器的内容则可以随心所欲地更改。
- 如果过程的返回值是 32 位或者更小位数的值，这些值将被返回到 EAX 中。如果是 64 位整数值，则返回到 EDX 和 EAX 中，其中低 32 位放在 EDX 中，高 32 位放在 EAX 中。如果是浮点型返回值，则返回到浮点堆栈的顶部。(在本书中，我不会涵盖浮点数的相关工作。)如果是字符串、结构或者其他位数大于 32 位的数据项，则通过引用返回，即，该过程返回一个指向它们的 32 位指针到 EAX 中。
- 传递给程序的参数被以相反的顺序压入堆栈中。也就是说，给定 C 函数 MyFunc(foo, bar, bas)，BAS 第一个被压入堆栈，bar 第二个被压入堆栈，foo 最后被压入堆栈。后面会有更多关于这方面的讲述。
- 过程本身并不从堆栈中移除参数。主调程序必须在过程返回之后做这件

事情。或者将过程从堆栈中弹出，或者(更为常见，因为它通常更快些)向堆栈指针寄存器 ESP 上添加一个偏移地址。(再次的，稍后我们真正实践这一点时将给出其含义的详细解释。)

- 虽然从技术上来讲不能被叫作调用约定，但是这个要求非常重要：命名汇编程序的起始点的标签必须是“main:”，而不是 start:，也不是 MAIN:，也不是 Beginning:。在代码开始处的标签必须是 main:，而且它必须被声明为全局的。

彻底理解了这些规则之后，你就能够调用标准 C 库以及许多其他非常有用的库中的众多函数了，所有这些库都是用 C 编写的，并且像前面描述那样遵循所有的公约。

12.2.2　建立一个框架

在程序中尊重 C 调用公约从保存并恢复前面已经列出的、被我称之为神圣的寄存器的内容开始。对于每一个连接到 glibc(或连接到几乎任何其他用 C 编写的函数库)的程序而言，其开始处和结束处的代码完全一样，因此，将所需的代码复制到一个样板文件中是非常有用的，这样，每当开始任何打算连接到 C 函数的新项目时，我们都可以使用它。

我已经创造了一个这种样板文件，名为 boiler.asm，它就在本书的清单存档中。为了保存和恢复必要的寄存器所需的代码都已经在那里了。本章所有其余的演示程序都基于 boiler.asm，你也可以将它作为自己的新项目的基础。作为一个简单的例子，你可以往后翻到清单 12-1 中的 eatclib.asm，它就是一个只添加了 5 行新代码的 boiler.asm 文件。

12.2.3　保存和恢复寄存器

如前所述，C 调用约定提供的不可思议的规定之一是：程序不得随意更改所有的通用寄存器。连接到 C 函数的 Linux 应用程序不能更改的寄存器有 EBX、ESP、EBP、ESI 和 EDI。你会发现，当程序开始时，boiler.asm 将这些寄存器保存到堆栈中，然后在将控制返回到 Linux 之前再恢复它们。

从保存 EBX、ESP、EBP、ESI 和 EDI 这一需求中，我们可以得出的一个非常重要，但却不很明显的结论：其他的通用寄存器可以被放入垃圾箱。是的，放入垃圾箱——而且不只被你放入垃圾箱。当你调用其他人编写的过程时，那些过

程也可能会改变 EAX、ECX 和 EDX 中的值。(堆栈指针 ESP 是一种特殊情况，需要某种不适用于其他寄存器的特殊照顾。) 这意味着当调用一个 C 库函数(如 printf())时，我们不能假设(例如)在 ECX 中跟踪的计数值能毫发无损地留下。

如果你正在使用 ECX 计算通过一个调用库函数或者不是你自己编写的函数的循环的次数，那么在调用库函数之前和从库函数返回之后，你都必须将 ECX 的值保存到堆栈上。这同样适用于 EAX 和 EDX。(EAX 通常用于返回库函数的值，所以，当进行库调用时，使用它来存储计数器的值和地址，并不是一个好主意。) 如果需要在一个库函数调用过程中交互地保存它们的值，必须在库函数被调用之前将其保存到堆栈上。

反之，EBX、EBP、ESI 和 EDI 的神圣的性质意味着当你进行 C 库调用时，这些寄存器将保持它们的值。对于你的约束力同样也是对于 C 库的约束力。那些必须使用这些寄存器的库函数会在不需你任何关注的情况下保存和恢复它们的值。

12.2.4　建立堆栈帧

堆栈在汇编语言工作中非常重要，而且在与 C 接口的程序中更是加倍重要，因为在 C(和实际上其他大多数本机代码高级语言，包括 Pascal)中，堆栈扮演着一个非常重要的角色。其中的原因很简单：编译器是写汇编语言代码的机器人，它们不是像你一样聪明的人。也就是说，一个编译器必须使用这些看似蛮力的方法来创建其代码，而这些方法严重依赖于堆栈的使用。

编译器生成代码是博士论文的内容，与这本入门级别的书所涉及的课程不是特别相关。在 Linux 上承担汇编语言编译工作的编译机制是堆栈帧机制。编译器依赖堆栈帧来创建函数的局部变量，虽然堆栈帧在纯粹的汇编工作中的用处很小，但是如果想调用某种高级语言编译器编写的函数，你必须了解它们。

堆栈帧是堆栈上的一个属于某一特定函数的位置。它基本上是指包含在两个寄存器——基址指针寄存器 EBP 和堆栈指针 ESP 寄存器——之间的位置。将其画出来可能比单调的解释更好些，见图 12-3。

堆栈帧是通过将主调程序的 EBP 寄存器的副本压入堆栈以将其保存，然后将主调程序的堆栈指针寄存器 ESP 拷贝到寄存器 EBP 中来创建的。任何履行了 C 调用公约的汇编程序的前两条指令都必须是这些：

```
push ebp
mov ebp,esp
```

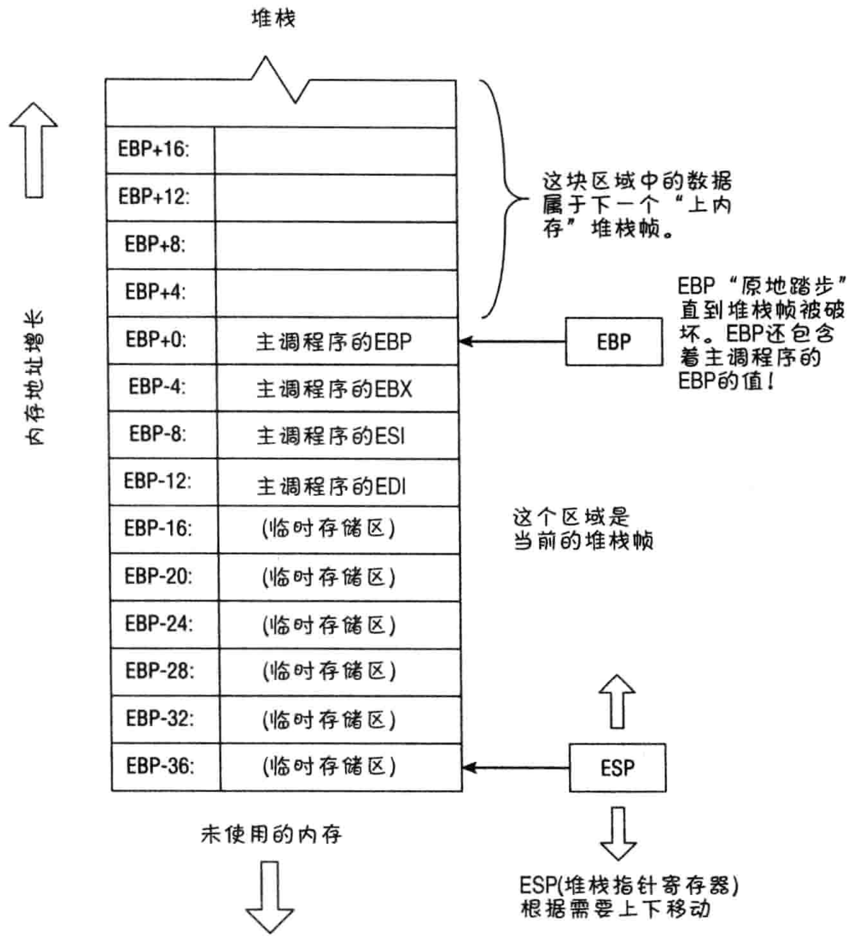


图 12-3 堆栈帧

在此之后，你必须或者不干预 EBP，或者，如果在紧要时刻必须使用它，在任何违背 C 库假设的改变发生之前，必须确保能够恢复它(我强力推荐不干预它！)。EBP 被认为是新堆栈帧的锚，这是不应该改变它的主要原因。在堆栈帧之上(即在更高的地址)将会存储一些往往需要在代码中引用的内容，EBP 是引用它们的唯一安全方法。(这些内容并没有显示在图 12-3 中，但是我会稍晚些返回它们。)

一个不太明显的事实是 EBP 也是一个能够藏匿主调程序的堆栈指针值(ESP)的“小洞”。这也是你一旦建立了堆栈帧就不能改变 EBP 值的另外一个理由。如

果在程序末尾返回控制时带出一个随机值到 ESP 中，肯定容易带来各种麻烦。

一旦 ESP 被作为堆栈帧的一端而安全地锚定后，堆栈指针寄存器 ESP 就可以根据需要自由地上下移动了。但是，你需要在堆栈上做的第一件事是将主调程序的 EBX、ESI 和 EDI 的值放入堆栈，如图 12-3 所示。当堆栈帧在程序结尾被销毁时，它们被从堆栈中弹回到主调程序(在我们的例子中，它是来自标准 C 库的启动/关闭代码)，它们的值与当启动代码将你的程序作为 main 函数调用时的值相同。

但是，一旦 EBX、ESI 和 EDI 已经就位，你就可以把任何需要暂存的内容压入和弹出堆栈了。调用 C 函数需要很多次压栈和出栈操作，这一点你很快就会看到。

12.2.5　销毁堆栈帧

程序在通过将控制返回给其启动/关闭代码(如果这种关系还是不很清楚的话，请参照图 12-2)，而结束执行之前，它的堆栈帧必须被销毁。这在很多人听来似乎是发生了一些错误，但事实并非如此：堆栈帧必须被销毁，否则你的程序就会崩溃。(“抛弃”可能是更好的“销毁”术语，但是程序员们喜欢丰富多彩的语言，随着你与他们接触的时间越来越多，你将会了解到这一点。)

在销毁堆栈帧并返回控制之前，我们必须清理堆栈。这意味着任何在程序运行期间可能压入堆栈的临时存储值必须消失。所有留在堆栈中的值应该是主调程序的 EBP、EBX、ESI 和 EDI 的值。从根本上来讲，如果 EDI 是我们必须保存到堆栈上的最后一个值，那么 ESP(堆栈指针寄存器)的值最好指向那个保存的 EDI 值，否则会有麻烦。

一旦清理干净堆栈，为了销毁堆栈帧，我们必须首先将主调程序的寄存器值弹回到相应的寄存器中，一定要确保这些弹出操作采用正确的顺序。将主调程序的 EBX 的值弹出到 EDI 中同样会使程序崩溃！既然那样，我们撤销在创造堆栈帧中所遵循的逻辑不就可以了吗？我们可以通过将 EBP 的值移入 ESP 来恢复主调程序的 ESP 值，最后再将主调程序的 EBP 值弹出堆栈：

```
mov esp,ebp
pop ebp
```

就这样！堆栈帧消失了，堆栈和神圣的寄存器现在和启动代码将控制权交给我们的程序时处于同一状态。现在它可以安全地执行将控制发送给 C 库关闭代码

的 RET 指令了。

12.2.6　通过 puts()输出字符

glibc 中最简单的有用函数是 puts()，它能够向标准输出发送字符。从汇编语言中调用 puts( )非常简单，仅需 3 行代码即可实现。清单 12-1 中的程序建立在文件 boiler.asm 的基础之上，我们能够在 main 程序的开始和结束处看到用于保存和恢复神圣的寄存器的样板代码。

清单 12-1：eatclib.asm

```
; 源文件名字       : EATCLIB.ASM
; 可执行程序名     : EATCLIB
; 版本             : 2.0
; 创建日期         : 10/1/1999
; 最后更新日期     : 5/26/2009
; 作者             : Jeff Duntemann
; 描述             : 演示了对于 glibc 中内容的调用，使用 NASM 2.05
;                    通过 puts()将一个简短的文本字符串发送到标准输出.
;
; 通过这些命令生成:
;    nasm -f elf -g -F stabs eatclib.asm
;    gcc eatclib.o -o boiler

[SECTION .data]             ; 包含已初始化数据的段

EatMsg: db "Eat at Joe's!",0

[SECTION .bss]              ; 包含未初始化数据的段

[SECTION .text]             ; 包含代码的段

extern puts                 ; 来自 glibc 的简单的"输出字符串" 例程
global main                 ; 这行代码是需要的，这样连接器才能找到入口点

main:
       push ebp             ; 为调试器建立堆栈帧
       mov ebp,esp
       push ebx             ; 必须保存 ebp、 ebx、 esi 和 edi
       push esi
       push edi
;;; 在此之前的所有内容都是样板代码，请在所有的一般应用程序中使用它!
```

```
        push EatMsg             ; 将消息的地址压入到堆栈中
        call puts               ; 调用 glibc 的用于显示字符串的函数，
        add esp,4               ; 清理堆栈，通过将 ESP 调回 4 个字节

;;; 在此之后的所有内容也是样板代码；请在所有的一般应用程序中使用它！

        pop edi                     ; 恢复已经保存的寄存器
        pop esi
        pop ebx
        mov esp,ebp                 ; 在返回之前，销毁堆栈帧
        pop ebp
        ret                         ; 将控制返回 Linux
```

调用 puts()的方法是一个关于调用 C 库例程一般过程的很好的小型范例，我们可以使用该方法调用几乎所有的 C 库例程。所有的 C 库例程都将它们的参数存放到堆栈上，这意味着我们必须或者压入长度为 32 位的数值，或者压入指向字符串或位于其他某处的某个较大的数据对象的指针。在本例中，我们压入一个 32 位的、指向某一文本字符串的指针。请注意，我们不是像通过 sys_write 内核调用发送文本到控制台上那样，传递该字符串的长度给 puts()。puts()从字符串的起始位置开始，将字符发送到标准输出，直到遇到一个 0(null)字符。位于该字符串的第一个字节和 0(null)字符之间的有多少个字符，控制台就接收多少个字符。

12.3 使用 printf()格式化文本输出

puts()库例程(library routine)可能看起来非常有用，但是与它的少数更先进的兄弟姐妹相比，它太小儿科了。通过 puts()，我们只能发送一个简单的文本字符串到一个文件(默认情况下，标准输出)，没有任何类型的格式。更糟糕的是，puts()始终在它的显示内容的结尾包含一个 EOL 字符，无论在字符串数据中是否包含该字符。这样我们就无法利用多次 puts()调用来在同一行输出多个文本字符串。

关于 puts()，我们充其量能够说它具有简单的美德。对于几乎所有的字符输出需求，我们最好使用一个功能更为强大的库函数——printf()。printf()函数使我们仅仅使用一个函数调用就能够完成很多真正有用的事情：

- 输出带有终止字符 EOL 的文本或者不带终止字符 EOL 的文本。
- 在输出数值型数据时，通过指定格式化代码，可将其转换成各种数字格

式的文本。

- 输出文本到一个包括多个分别储存的字符串的文件。

如果已经使用 C 工作了半小时以上，那么 printf()对你来讲会特别清楚，但是对于那些来自其他语言的人们(如 Pascal，它没有直接的对应内容)而言，它可能需要一些解释。

printf( )例程很高兴显示一个简单的字符串，如“到乔的饭馆去吃饭！”，——但是，当将字符串输出到标准输出时，我们也可以合并其他的文本字符串、对基字符串中的数值型数据进行格式转换，并能将所有内容无缝地显示在一起。这是通过在基字符串中放入一些格式化代码，然后与这个基字符串一起，为每一个这些格式化代码传递一个数据项到 printf( )中来实现的。格式化代码使用百分号开头，并且包括与合并到基字符串中的数据项的类型和大小有关的信息，以及如何展现这些信息的相关内容。

让我们从观察一个非常简单的例子入手。这里有一个包含一个格式化代码的基字符串：

```
"The answer is %d, and don't you forget it!"
```

%d 格式代码告诉 printf()，有一个有符号整数值需要被转换为文本，并且用该文本替换基字符串中相应的格式化代码。当然，你现在必须传递一个整数值到 printf()中，稍后我会告诉你如何做这件事情，但是当你这样做的时候，printf()将会把该整数转换成文本，并当发送文本到流中时，将其与基字符串合并。如果传递的是十进制值 42，在控制台上将会看到这样的结果：

```
The answer is 42, and don't you forget it!
```

格式化代码实际上有很多种结构，并且 printf()机制作为一个整体来讲，有比我在这里描述的多得多的技巧。任何一个优秀的 C 参考资料都会详细地解释全部内容。表 12-1 列出了最常见也最有用的格式化代码。

表 12-1 常用的 printf()格式化代码

格式化代码	基字符串	描述
%c	n/a	输出单个字符
%s	n/a	输出字符串
%x	16	转换一个整数，并将其以十六进制显示
%%	n/a	显示一个百分号符号
%d	10	转换一个整数，并将其以十进制显示

我们可以对格式代码做得最显著的增强是在%符号和代码字母之间放入一个整数值：

```
%5d
```

此代码告诉 printf()，右对齐显示一个 5 个字符宽的值。如果这个值没有包含宽度值，那么 printf()将按它的实际宽度来显示它。

请记住，如果需要显示一个百分号符号，我们必须在字符串中包含两个连续的百分号符号：第一个是格式化代码，告诉 printf()将第二个百分号符号显示为它自己，而不是作为一个格式化代码的“引线”。

向 printf()传递参数

假设你已经在逻辑上了解了 printf()的工作原理，使用它的真正挑战在于知道如何向它传递为了处理任意特定字符串显示所需的所有参数。像 Pascal 中的 Writeln()函数一样，printf()没有固定数量的参数。它可以采用非常少的参数，以至于可以只需要一个基字符串作为参数，它也可以采用许多参数，你需要多少，就采用多少，其中包括许多附加的字符串、字符值以及各种类型的数字值。

所有 C 库函数的参数都传递到堆栈上。这可以采用通过将参数值本身压入堆栈的直接方式完成，也可以采用通过将参数的 32 位地址引用压入堆栈的间接方式来完成。对于 32 位或 64 位数据值而言，我们将参数本身压入堆栈。而对于更大的数据项而言，如字符串和数组，我们则把指向该项的指针压入堆栈。(在 C 的术语中，传递某些数据的地址叫作传递指向该数据的指针。)

当多个参数被传递给 printf()后，它们都必须被压入堆栈，并且采用一种非常特殊且有违直观的压栈顺序：从右到左，就像当你从 C 代码中调用 printf()时它们将出现的顺序一样。基字符串被认为是最左边的参数，始终最后一个被压入堆栈。这里有一个来自于 C 的简单示例，它将会有助于你的理解：

```
printf('%d + %d = %d ...for large values of %d.',2,2,5,2);
```

这是一个调用 printf()的 C 语言语句。该基字符串被圈在单引号中，它是第一个参数。基字符串后面是几个数字参数。对于每一个嵌入该字符串%d 格式化代码，都必须有一个数字值与之对应。这些项进入堆栈的顺序必须为从右边开始，向左侧依次读取：2，5，2，2，基字符串最后一个被压入堆栈。在汇编语言里，它是这样做的：

```
push 2
push 5
push 2
push 2
push mathmsg
call printf
add esp,20
```

标识符 mathmsg 是基字符串，它的地址是所有参数中最后一个被压入堆栈的。请记住，我们并不将字符串本身压入堆栈。我们压入的是该字符串的地址，C 库代码将使用它自己的机制，沿着这个地址来获取该字符串的数据。

序列末尾的 ADD 指令代表后面你将听到的名为"清理堆栈"的操作。每当通过一个 PUSH 指令将某些内容压入堆栈时，堆栈指针寄存器 ESP 都会向低内存移动一定数量的、与压入堆栈内容大小相等的字节。在这里的示例中，所有参数都是完全相同的四个字节大小。由于有 5 个这样的参数，因此为了进行调用，ESP 中的值需要改变 20 个字节。在调用完成后，ESP 必须回到当我们开始压入参数到堆栈之前，它原来所在的地方。通过将 ESP 中的值增加 20，堆栈指针往高内存方向移回了 20 个字节，这样它就回到了我们开始设置 printf()调用时，它原来的位置。

一定要注意：如果你忘记了清理堆栈，或者在你清理堆栈时将指针向上移动了错误数量的字节数，你的程序几乎肯定会引发分段错误。细节——我敢称之为整洁吗？——很重要！

将几个文本字符串合并成基字符串或多或少采用相同的方式来完成，只不过使用的格式代码是%s，而不是%d：

```
push dugongs            ; 最右侧的参数最先被压入堆栈
push mammals            ; 压入左侧的下一个参数
push setbase            ; 基字符串被最后压入堆栈
call printf             ; 进行 printf()调用
add esp,12              ; 堆栈清理: 3 个参数 x 4 个字节 = 12
```

printf()调用所需的字符串在程序的.data 段中进行定义：

```
MammalMsg db 'Does the set of %s contain the set of %s?',10,0
Mammals db 'mammals',0
Dugongs db 'dugongs',0
```

为了简洁起见，我没有在这里显示整个程序——那样的话，得需要好长时间

才能看完所有的样板程序——但是到现在为止，你应该能够具备对 printf()进行调用的意识。请记住以下三个关键方面：

- 参数被由右至左压入堆栈，从函数调用开始，像它们在 C 语言中的写法一样。基字符串被最后一个压入堆栈。如果你正在做的事情需要稍微复杂地使用 printf()，将其先以 C 的形式写出，然后再翻译成汇编语言会对你有所帮助。首先用 C 编写一个简短的程序来测试 printf()的调用，然后编译、运行它，这会对你有更大的帮助，特别是当你正在编写一些拥有很多字符串和格式化代码的大型程序时。
- 当调用 printf()之后，我们必须将 ESP 的值和压入堆栈的所有参数的字节数的总和相加。不要忘记，对于字符串而言，我们压入堆栈的是该字符串的地址，而不是包含在其中的数据！对于几乎所有的参数而言，这将是 4 个字节。对于 64 位整数而言，它将是 8 个字节。
- printf()函数调用能够破坏除了神圣的寄存器之外的一切内容。在整个 printf()调用中，不要指望将值保存到其他寄存器中！例如，如果我们在执行一个调用 printf()的循环期间，试图将一个计数器值保持 ECX 中，那么 printf()调用将会破坏 ECX 中的计数值。我们必须在每次进行库函数调用之前，将 ECX 保存到堆栈上，并且在库调用返回之后将其恢复——或者，尽最大努力尽可能小心地使用一个可用的神圣寄存器，如 ESI、EDI 或者 EBX 寄存器。(显然，不要尝试使用 ESP 或 EBP！)

我们应该小心地确保所有字符串参数都正确地以一个二进制 0 终止，这是唯一一种诸如 puts()和 printf()之类的 glibc 函数知道字符串数据从哪里结束的方式。再次地，如果 printf()调用不能在汇编语言中正常工作，你可以编写一个简单的只包含 printf()调用的单行 C 程序，看看它能否在那里奏效，并且按照预期的方式工作。如果确实能够正常工作的话，你可能在汇编程序中把参数的顺序或者参数的数量搞错了。永远不要忘记，对于每个格式代码，都必须对应一个参数。

本书的代码清单存档中包含一个完整的示例程序——answer.asm，它显示了所有本节讲述的 printf()调用。

12.4　使用 fgets()和 scanf()进行数据输入

使用 INT80h 和 sys_read 内核调用从 Linux 键盘读入字符非常简单，但不是很灵活。标准 C 库中有一种更好的方式。事实上，从键盘(它是 C 库分配给标准

输入的默认数据源)中读入数据的 C 库函数几乎与那些显示数据到标准输出的 C 库函数正好相反。这是故意的，尽管有些时候，这种对称性也会碍事，我将会在稍晚些给出解释。

如果在 C 库参考资料中“闲逛”(你应该这样做——那里有很多有趣的例程，你可以从汇编程序中调用它们)，你会发现 gets()例程。你可能想知道(如果我没有选择在这里告诉你)我为什么涵盖它。gets()例程本身很简单：如果向它传递一个打算存放字符的字符串数组的名字，然后用户从键盘输入字符，那么这些字符将被放入该数组中。当用户按下 Enter 键时，gets()在输入文字的末尾追加一个 null，并返回。那么，是什么令人不悦呢？

嗯，这个数组有多大呢？还有，用户有多笨呢？这里是隐藏的困难：没有办法告诉 gets()何时停止接收字符。如果用户输入的字符比分配给数组的空间还要多，gets()将会兴高采烈地继续接受字符，覆盖掉所有位于该数组旁边的内存中的任何数据。如果那是某些重要的内容，你的程序几乎肯定会发生故障，并且很可能崩溃。

这就是为什么当你尝试使用 gets()时，gcc 会警告你 gets()是危险分子的原因。它已经很古老了，在 Unix 和标准 C 库首次设计之后的几十年中，已经有更好的机制被创造出来。gets()指定的继承者是 fgets()，它有一些内置的安全机制——同时也有一些复杂性。

复杂性源于这样的事实：你必须向 fgets()传递一个文件句柄。一般来说，标准 C 库中以 F 开头的例程都与文件有关。(本章稍后部分将讲解如何使用磁盘文件。) 我们可以使用 fgets() 从一个磁盘文件中读取文本——但是请记住，在 Unix 术语中，键盘已经被连接到一个名为“标准输入”的文件上。如果能够将 fgets() 连接到标准输入上，我们就可以从键盘阅读文本，而这正是自动完成古老而危险的 gets ()函数所做的事情。

使用 fgets()的好处是我们能够为例程指定接收来自键盘的字符的最大个数。用户输入的任何其他内容将被截断并丢弃。如果这个最大值不比你定义的用于存放用户输入的字符的字符串缓冲区大，使用 fgets() 就没有机会让程序崩溃。

将 fgets()与标准输入相连非常容易。正如本书前面部分中讲的那样，Linux 预定义三个标准的文件句柄，它们被自动连接到你的程序：stdin(标准输入)、stdout(标准输出)和 stderr(标准错误)。为了通过 fgets()从键盘接受输入，我们要使用 stdin 标识符。它已经存在，为了从汇编语言内部引用它，我们只需简单地将其声明为 extern。

以下是 fgets()函数的使用方法：

(1) 确保在程序的.text 段顶部已经(与其他外部声明一起)声明了 extern fgets 和 extern stdin。
(2) 在程序的.bss 段通过使用 RESB 指令在程序的.bss 段声明一个足够大的缓冲区变量，以保存你想让用户输入的字符串数据。
(3) 为了调用 fgets()，首先将文件句柄压入堆栈。注意：必须压入句柄本身，而不是句柄的地址！因此，使用这种格式：push dword [stdin]。
(4) 把用于指明想要 fgets()接收的最大字符数的值压入堆栈。一定要确保它不比你在.bss 段中定义的缓冲区变量大！堆栈必须包含实际的值——不要压入存放这个值的变量的地址。将一个立即数或者一个内存变量的内容压入堆栈将会奏效。
(5) 把 fgets()即将用于存储用户输入的字符的缓冲区变量的地址压入堆栈。
(6) 调用 fgets() 本身。
(7) 像调用所有用 C 语言编写的库函数一样，不要忘了清理堆栈！

实际情况可能比听起来要好一些。在实际的代码中，对 fgets()的调用应该类似以下内容：

```
push dword [stdin]     ; 将预定义的标准输入的文件句柄压入堆栈
push 72                ; 从键盘接收不多于 72 个字符
push InString          ; 将用于存放输入字符的缓冲区的地址压入堆栈
call fgets             ; 调用 fgets()
add esp,12             ; 清理堆栈: 3 个参数 X 4 个字节 = 12
```

这里，标识符 InString 是一个内存变量，定义如下：

```
[SECTION .bss]         ; 包含未初始化数据的段
InString resb 96       ; 为字符串入口缓冲区保留 96 个字节
```

回想一下，RESB 指令的作用只是为变量预留空间。该空间并没有使用任何特定值——空格、空值或任何其他值——预先清除。在用户通过 fgets()输入数据之前，使用 RESB 分配的字符串存储空间都是未初始化的，其中可以包含任何垃圾值。它一般情况下充满了空值，但是 Linux 并不向你做任何承诺！

从用户端的屏幕上，fgets()只是简单地接受字符，直到用户按下 Enter 键。当用户敲入最大允许的字符数后，它并不自动返回。(以防用户使用 Backspace 键回退输入并更正它。)但是，用户输入的任何超出最大字符数允许的内容将被丢弃。

后面的清单 12-2 所示的 charsin.asm 文件包含了前面的代码。

使用 scanf() 输入数字值

以一种特殊的方式，C 库函数 scanf( )与 printf( ) 刚好相反：不是以字符流的形式输出一个输出格式化数据，scanf ( ) 函数能够从键盘上接收字符数据流，并能将其转换为数字数据，存储在数字型变量中。 scanf()工作得非常好，并且它能理解很多本书没有讲解的格式，尤其是对于输入浮点数的情况。(浮点值是汇编语言编程中的一种特殊问题，我在这一版中不会涉及它们。)

对于大多数可以从容编写的简单程序而言，我们一般会输入简单的整数，scanf()非常擅长做这件事情。它的用法很简单：传递给 scanf 函数存放输入值的数字变量的名称，以及一个说明数据输入将采用哪种形式的格式代码。scanf()函数接收由用户输入的字符，并将其转换为该字符代表的整型值。例如，scanf 函数()能够读入两个连续进入的 ASCII 字符“4”和“2”，并且当用户按回车键之后，将其转换为以 10 为基数的数字值 42。

那么，对于指导用户输入什么内容的提示字符串呢？嗯，很多汇编新手假设可以将提示字符串与一个格式代码结合在一个简单的字符串中，然后将其传给 scanf()——但是很可惜，这将无法工作。似乎它应该工作——毕竟，我们可以将格式代码与即将使用 printf()显示的基字符串结合在一起。在 scanf()函数中，从理论上来讲，我们可以使用一个包含格式代码的基字符串，但是用户将不得不键入该提示字符串以及数字数据！

实际上，scanf()函数使用的唯一字符串是一个包含格式代码的字符串。如果想显示一个提示信息，必须在调用 scanf()之前，使用 printf()将其显示出来。为了保持提示信息和录入的数据位于同一行，一定要确保不要在提示字符串的结尾包含 EOL 字符。

scanf()函数自动从标准输入中接收字符。没有必要像使用 fgets()函数那样向它传递标准输入文件的句柄。有一个单独的 glibc 函数，fscanf()，使用它时，你倒是必须向它传递一个文件句柄，但是对于整数数据输入而言，使用 scanf 函数()没有任何危险。

这里是 scanf()例程的使用方法：

(1) 确保已经在.text 的顶部，与其他外部声明一起，声明了 extern scanf。
(2) 声明一个正确类型的内存变量用来保存 scanf()要读取和转换的数字数据。我在这里的例子是为整数数据而列举的，所以要使用 DD 指令或

RESD 指令创建一个这样的变量。显然，如果打算保存若干个单独的值，你需要为每一个输入的值声明一个变量。

(3) 为了调用 scanf()以输入一个单个值，首先将用来存放该值的内存变量的地址压入堆栈。(关于如何在一次调用中输入多个值，参见下面的讨论。)

(4) 把用以指定数据将以什么格式进入的格式字符串的地址压入堆栈，对于整型值而言，这通常是字符串%d。

(5) 调用 scanf()。

(6) 清理堆栈。

一个典型的 scanf( )调用代码是这样的：

```
push IntVal     ; 将整型缓冲区的地址压入堆栈
push Iformat    ; 将整型格式字符串的地址压入堆栈
call scanf      ; 调用 scanf 来输入数字数据
add esp,8       ; 清理堆栈：2 个参数 X 4 个字节 = 8
```

我们也可以向 scanf()提供含有多个格式代码的字符串，以便用户可以通过一次 scanf()调用就能输入多个数字值。我已经尝试过这一点，它将导致一种非常独特的用户界面。如果你正在编写一个程序，用来读取一个包含用多行文本方式表达的整数值的文本文件，并将其转换为内存中实际的整数变量，这一属性将得到更好地的应用。如果只是简单地通过键盘从用户那里获得数字值，最好每次调用 scanf() 时只接受一个值。

清单 12-2 中的 charsin.asm 程序展示了对于那些既能接收来自用户从键盘输入的字符串数据，也能接收数字数据的应用，如何在输入域的旁边设置提示信息。在接收数据之后，程序会通过 printf( )将输入的内容显示出来。

清单 12-2：charsin.asm

```
; 源代码文件      : CHARSIN.ASM
; 可执行程序名    : CHARSIN
; 版本            : 2.0
; 创建日期        : 11/21/1999
; 最后更新日期    : 5/28/2009
; 作者            : Jeff Duntemann
; 描述            : 一个 Linux 下的字符输入演示程序，使用 NASM 2.05,
;                   结合了 fgets()调用和 scanf()调用.
;
; 采用以下命令生成:
;    nasm -f elf -g -F stabs charsin.asm
```

```
;    gcc charsin.o -o charsin
;

[SECTION .data]             ; 包含已初始化数据的段

SPrompt db 'Enter string data, followed by Enter: ',0
IPrompt db 'Enter an integer value, followed by Enter: ',0
IFormat db '%d',0
SShow db 'The string you entered was: %s',10,0
IShow db 'The integer value you entered was: %5d',10,0

[SECTION .bss]              ; 包含未初始化数据的段

IntVal resd 1               ; 保存一个未初始化的双字
InString resb 128           ; 为存储输入字符串数据的缓冲区预留 128 个字节，

[SECTION .text]             ; 包含代码的段

extern stdin                ; 用于输入的标准文件变量
extern fgets
extern printf
extern scanf
global main                 ; 该行代码是必需的，这样连接器才可以找到入口点

main:
      push ebp              ; 为调试器建立堆栈帧
      mov ebp,esp
      push ebx              ; 程序必须保存 ebp、ebx、esi 和 edi
      push esi
      push edi
;;; 在此之前的所有内容都是样板代码；请在所有的一般应用程序中使用它！
; 首先，是一个安全地局限于字符串输入的 fgets()使用示例：
      push Sprompt          ; 将提示字符串的地址压入堆栈
      call printf           ; 显示它
      add esp,4             ; 为 1 个参数做堆栈清理
      push dword [stdin]    ; 将标准输入的文件句柄压入堆栈
      push 72               ; 从键盘接收不超过 72 个字符
      push InString         ; 将用于存放输入字符的缓冲区地址压入堆栈
      call fgets            ; 调用 fgets
      add esp,12            ; 堆栈清理：3 个参数*4 个字节 = 12
```

```
        push InString       ; 将存放输入字符串数据的缓冲区地址压入堆栈
        push Sshow          ; 把用于字符串显示的提示信息的地址压入堆栈
        call printf         ; 显示它
        add esp,8           ; 堆栈清理: 2 个参数*4 个字节 = 8
;下一步, 使用 scanf() 来输入数字数据:
        push Iprompt        ; 将用于整数输入的提示信息的地址压入堆栈
        call printf         ; 显示它
        add esp,4           ; 为 1 个参数做堆栈清理工作

        push IntVal         ; 将整数缓冲区的地址压入堆栈
        push Iformat        ; 将整数格式字符串的地址压入堆栈
        call scanf          ; 调用 scanf 输入数字数据
        add esp,8           ; 堆栈清理: 2 个参数*4 个字节 = 8

        push dword [IntVal] ; 将打算显示的整数值压入堆栈
        push Ishow          ; 压入基字符串
        call printf         ; 调用 printf 转换并显示这个整数
        add esp,8           ; 堆栈清理: 2 个参数*4 个字节 = 8
;;; 在此之后的所有内容都是样板代码; 请在所有的一般应用程序中使用它!
        pop edi             ; 恢复已保存的寄存器
        pop esi
        pop ebx
        mov esp,ebp         ; 在返回之前, 销毁堆栈帧
        pop ebp
        ret                 ; 将控制返回给 Linux

[SECTION .data]             ; 包含已初始化数据的段

sprompt    db 'Enter string data, followed by Enter: ',0
iprompt    db 'Enter an integer value, followed by Enter: ',0
iformat    db '%d',0
sshow      db 'The string you entered was: %s',10,0
ishow      db 'The integer value you entered was: %5d',10,0

[SECTION .bss]              ; 包含未初始化数据的段

intval resd 1               ; 保存一个未初始化的双字
instring resb 128           ; 为存储输入字符串数据的缓冲区保留 128 个字节
```

上述演示程序的一个缺点是：没有对输入数字进行任何验证。如果用户输入

的代表一个数字值的 ASCII 值太大，以至于无法包含在一个 32 位整数中，或者输入了某些不能代表一个数字值的字符混合物，scanf()函数返回给该程序的值将是一个与用户输入信息没有必然联系的垃圾数据。

12.5 驾驭时间

标准 C 库包含一组数量众多的能够操作日期和时间的函数。虽然这些函数原本是为了处理由 20 世纪 70 年代很流行的古老的 AT&T 微型计算机硬件实时时钟产生的日期值而设计，现在它们已经变成了一个到任一操作系统实时时钟支持的标准接口。为 Windows 编写 C 程序的人们也使用同样的一组函数，他们采用与正在使用的操作系统无关的或多或少相同的工作方式。

通过了解如何像汇编语言程序过程一样调用这些函数，我们可以读取当前日期、以各种各样的格式表达时间和日期值、对文件应用时间戳，以及做很多其他非常有用的事情。

让我们看看它是如何工作的。

12.5.1 C 库的时间机制

在标准 C 库内部深处的某个地方，有一个代码块，当被调用时，该代码块能够查看计算机的实时时钟，读取当前日期和时间，并将其转换成一个标准的 32 位无符号整数值。这个值（理论上来讲）是已经在(始于 1970 年 1 月 1 日 00:00:00 通用时间的)“Unix 时代”度过的秒数。每度过 1 秒，该值就会加 1。当通过 C 库读取当前时间或日期时，我们检索的是这个数字的当前值。

这个数字叫作 time_t。time_t 值于世界标准时间(UTC)2001 年 9 月 9 日上午 07 时 46 分 40 秒翻转到了 10 位(自 1970 年 1 月 1 日以来的 10 亿秒)。在不久的将来暂时还没有千年虫式的危险，但是在 2038 年 1 月 19 日上午 03 点 14 分 07 秒，将 time_t 作为一个有符号整数处理的计算机将会看到它翻转到 0，因为一个 32 位有符号整数能够表达的最大量为 2 147 483 647。这是很多秒(可以有一个相当长的时间去做准备)，但是那时我将达到 86 岁高龄，希望它发生时我仍然健在。

不用担心。一个正常实现的 C 库根本不认为 time_t 是一个 32 位的量，所以当有符号 32 位 time_t 翻转到 2038 年时，我们将已经至少使用 64 位值来表示一切内容，整个问题将推迟到另外一个 292 亿年左右。如果我们到那时还没有一劳

永逸地修复它，那么我们将值得进入宇宙学家们正在预测的不久之后的宇宙压缩中。

一个 time_t 值仅为一个任意的秒计数，它并不能告诉我们多少关于它自己的信息，尽管它在以秒为单位计算已经流逝的时间方面很有用。另一种由标准 C 库实现的标准数据类型要更有用些。tm 结构(通常被叫作一个结构，而在 Pascal 中被叫作记录)是一组 9 个 32 位的数字值，分别以不同的有用块表达当前时间和日期，如表 12-2 中的总结所示。请注意，虽然一个结构(或记录)名义上是一个不同值的分组，但在当前的 x86 Linux 实现中，一个 tm 值更像是一个数组或者一张数据表，因为所有 9 个元素的大小都相同，即 32 位或者 4 个字节。通过为每一个元素包含一个相对于结构开始位置的偏移值，我已经以这种方式在表 12-2 中对其进行了描述。这使得我们能够使用一个指针指向该结构，并对从结构开始到结束范围内的任一给定元素使用偏移值。

表 12-2　包含在 tm 结构中的值

以字节为单位的偏移值	C 库名称	定义
0	tm_sec	分钟后面的秒，从 0 开始
4	tm_min	小时后面的分钟，从 0 开始
8	tm_hour	日期后面的小时，从 0 开始
12	tm_mday	月中的某一天，从 1 开始
16	tm_mon	年中的某一月，从 0 开始
20	tm_year	自 1900 年以来的年份，从 0 开始
24	tm_wday	自星期日以来的天数，从 0 开始
28	tm_yday	某年中的第几天，从 0 开始
32	tm_isdst	夏令时间标志

有一些 C 库函数能够将 time_t 值转换为 tm 值，并能再转换回来。本章将涵盖一些，它们都非常简单，一旦已经彻底消化了 C 调用公约，理解对于它们的任何汇编调用协议将不是问题。

请注意，time_t 值并非真的是自 Unix 时代开始的秒数的精确值。在 Unix 进行秒计数的方式中存有一些故障，time_t 并不能像现实世界中的 NIST 时间那样对累积天文错误进行调整。因此，在很短的间隔(理想情况下，少于一年)内，time_t 可被认为是准确的。除此之外，假设它关闭几秒钟或者更多的时间，将没有简单方法能够算出如何弥补那些错误。

12.5.2　从系统时钟中取出 time_t 值

在 Unix 兼容系统中，可用 32 位无符号整数来表示任意单个时间的秒数(至少是在 1970 年 1 月 1 日之后的)。取出当前时间值可以通过调用 time()函数实现：

```
push dword 0              ; 将一个 32 位空指针压入堆栈，我们不需要缓冲区
                          ; 时间值被返回到 eax 中
call time                 ; 将日期值返回到 eax 中
add esp, byte 4           ; 为一个参数清理堆栈
mov [oldtime],eax         ; 将时间值保存到内存变量中
```

time() 函数可能将 time_t 值返回到两个地方：到 EAX 中或者到某处为其分配的缓冲区中。为了让 time()将值放到缓冲区中，我们需要在堆栈上给它传递一个指向该缓冲区的指针。如果不想将该 time_t 值存储到缓冲区中，我们仍然必须在堆栈上给它传递一个空指针。这就是为什么我们在前面的代码中把一个 0 值压入堆栈的原因：0 是一个空指针的值。

不需再传递给 time()其他参数。当它返回时，在 EAX 中将拥有当前的 time_t 值。就这么简单。

12.5.3　将 time_t 值转换为一个格式化字符串

time_t 值本身并不能告诉我们很多内容。C 库中包含一个名为 ctime()的函数，该函数能够返回一个指向某一给定 time_t 的格式化字符串表达形式的指针。它返回一个指向埋藏在运行时库深处的字符串的指针。该字符串的格式如下：

```
Wed June 3 12:13:21 2009
```

第一个字段是由一个由 3 个字符组成的代码，代表星期几，后面是另外一个由 3 个或 4 个字符组成的代码，代表月份，再后面是一个占用两个位置的字段，用于表示某月中第几天。再往后是以 24 小时格式表示的时间，最后结尾的是年份。另外(尽管有时它是一个滋扰)，作为 ctime()返回值的字符串以一个换行符字符终止。

这里是 ctime( )的调用方法：

```
push OldTime    ; 将日历时间值的地址压入堆栈
call ctime      ; 返回一个指向 ASCII 时间字符串的指针到 eax 中
add esp,4       ; 为一个参数进行堆栈清理
```

这看起来很普通，但是请注意,这里的某些内容背离了最近的 glibc 使用经验：传递给 time()的是一个 time_t 值的地址，而不是该 time_t 值本身！我们已经习惯了通过将 32 位整数值本身压到堆栈上以传递该 32 位整数值——比如说，通过 printf()将其进行显示。这里不是这样。目前，time_t 值在 Linux 下用一个 4 字节整数表示，但是也不能保证永远是这样。因此，为了使其选项保持开放(并确保 Unix 可被使用数千年甚至数十亿年，天哪)，C 库需要一个指向当前时间的指针，而不是该时间值本身。

将一个指向打算以字符串形式表示的 time_t 值的指针压入堆栈之后，我们就可以调用 ctime()了。ctime()会返回一个指向保存在库内部某处的字符串的指针。通过 printf()函数，我们能够使用那个指针在屏幕上显示该字符串，或将其写入一个文本文件中。

12.5.4　生成单独的本地时间值

C 库还提供了一个函数，它能够将组成日期和时间的各个部分分割为单独的值，以便于我们单独使用它们或者采用不同的组合使用它们。这个函数就是 localtime()，只要给定一个 time_t 值，它就会将其分割成 tm 结构的各个字段，如表 12-2 所描述的那样。这里是它的调用代码：

```
push OldTime            ; 将日历时间值的地址压入堆栈
call localtime          ; 返回指向静态时间结构的指针到 eax 中
add esp,4               ; 为 1 个参数清理堆栈
```

在这里，OldTime 是一个 time_t 值。给定了这个值，localtime()就能够返回到 EAX 中——以非常类似于 ctime()的方式——一个指向 tm 结构的、位于 C 库内部某处的指针。通过将这个指针作为基址，我们可以访问结构中的各个字段。诀窍在于知晓你想要的单独的时间/日期字段到 tm 的偏移值，并以此作为一个从基址处开始的常量位移量。例如，下面的代码将使用 printf 在控制台上显示一个 tm 结构的年字段。请注意，第一条指令到 tm 的索引为 20 个字节，这是从结构开始位置到年字段的偏移量：

```
mov edx, dword [eax+20]   ; 年份的值在 tm 中的偏移量是 20 个字节
push edx                  ; 将该值压入堆栈
push yrmsg                ; 将基字符串的地址压入堆栈
call printf               ; 用 printf 显示字符串和年份的值
add esp, byte 8           ; 堆栈清理: 2 个参数 X 4 个字节 = 8
```

通过使用如表 12-2 所示的位移，我们可以访问 tm 结构中的其他所有时间和日期组成部分，并将每一个都存储为一个 32 位的整数值。

12.5.5 通过使用 MOVSD 复制 glibc 的 tm 结构

有时保存一份单独的 tm 结构的副本会很方便，尤其是当我们一次需要处理几个日期/时间值时。所以，当使用 localtime()来用日期/时间值填充 C 库的隐藏 tm 结构时，我们可以将那个结构复制到一个在程序的.bss 段或者.data 段中分配的结构中。

要想做这样一份副本，我们需要直接使用第 11 章中介绍过的 REP MOVSD (Repeat Move String Double)指令。MOVSD 几乎可以说是一条神奇的指令：一旦设置了指向打算拷贝的数据的指针和想要将它复制到的目标位置，并将该区域的大小存入 ECX 中，REP MOVSD 就能够完成其余的事情了。它在一次操作中就能将整个缓冲区的内容从内存中的一个地方复制到另一个地方。

要想使用 REP MOVSD，我们必须将源数据的地址——也就是说，即将被复制的数据——放入 ESI 中；将目标位置的地址——数据即将被放入的位置——放入 EDI 中；将即将被移动的项的数量放入 ECX 中。然后，确保方向标志 DF 被清零(关于这方面的更多内容，请参照第 11 章)，并执行 REP MOVSD：

```
mov esi,eax          ; 将静态 tm 的地址从 EAX 复制到 ESI 中
mov edi,tmcopy       ; 将本地 tm 副本的地址放入 edi 中
mov ecx,9            ; 在 Linux 下，一个 tm 结构的大小是 9 个双字
cld                  ; 将 df 清零，以便于我们采用上内存的方式移动
rep movsd            ; 将静态 tm 结构复制到本地副本中
```

这里，我们正在将 C 库的 tm 结构移到一个在程序的.bss 段中分配的缓冲区中。tm 结构是 9 个双字——36 个字节——的大小，所以我们必须要保留这么多的空间，并给它取一个名字：

```
TmCopy resd 9       ; 为时间结构 tm 保留 9 个 32 位字段
```

上述代码假定 EAX 中存放的是 C 库的已经填写了数据的 tm 结构的地址，并且已经为一个名为 Copytm 的 Tm 结构分配了内存。一旦执行，它将会把所有 tm 数据从它们的位于 C 运行库内部的藏身之所复制到刚刚分配的缓冲区中。

如第 11 章所讲的那样，rep 前缀把 MOVSD 设为自动机枪模式。也就是说，MOVSD 将继续从 ESI 指定的地址处移动数据到 EDI 指定的地址中，每移动一

次，ECX 计数值递减 1，直到 ECX 中的值变为零。然后它停止。

一个经常发生的错误是忘记 ECX 中的计数值是即将被移动的数据项的个数，而不是即将被移动的字节数！凭借助记符结尾的 D，MOVSD 指令可以移动双字，放入 ECX 中的值必须是即将被移动的大小为 4 个字节的项的个数。因此，在移动 9 个双字的过程中，MOVSD 实际上从一个位置传输了 36 个字节到另外一个位置——但是你是按双字而不是按字节来进行计数的。

清单 12-3 中的程序将所有这些代码片断组织在一起，形成了一个关于 Unix 的主要时间属性演示程序。还有很多 C 库时间函数有待研究，根据现在已经了解的关于 C 函数调用的相关知识，我们应该能够推算出对于任何其他函数的调用协议。

清单 12-3：timetest.asm

```
; 源文件名字      : TIMETEST.ASM
; 可执行程序名    : TIMETEST
; 版本            : 2.0
; 创建日期        : 12/2/1999
; 最后修改日期    : 5/26/2009
; 作者            : Jeff Duntemann
; 描述            : 一个为 Linux 编写的与时间函数相关的演示程序，使用 NASM 2.05
;
; 使用以下命令生成:
;     nasm -f elf -g -F stabs timetest.asm
;     gcc timetest.o -o timetest
;

[SECTION .data]            ; 包含已初始化的数据的段

TimeMsg    db "Hey, what time is it? It's %s",10,0
YrMsg      db "The year is %d.",10,0
Elapsed    db "A total of %d seconds has elapsed since program
   began running.",
10,0

[SECTION .bss]             ; 包含未始化数据的段

OldTime resd 1             ; 为时间值保留 3 个整数(双字)
NewTime resd 1             ; 32 位时间值
TimeDiff resd 1            ; 32 位时间值
TimeStr resb 40            ; 为时间字符串保留 40 个字节
```

```
TmCopy resd 9              ; 为时间结构保留 9 个整数字段

[SECTION .text]            ; 包含代码的段

extern ctime
extern difftime
extern getchar
extern printf
extern localtime
extern strftime
extern time

global main                ; 这条指令是必需的，以便连接器能够找到入口点

main:
       push ebp            ; 为调试器建立堆栈帧
       mov ebp,esp
       push ebx            ; 程序必须保留 EBP、 EBX、 ESI 和 EDI
       push esi
       push edi
;在此之前的所有内容都是样板代码; 请在所有的一般应用程序中使用它!
; 使用 glibc 的 time()函数产生一个 time_t 日历时间值:
       push 0              ; 将一个 32 位空指针压入堆栈，因为我们不需要缓冲区，
       call time           ; 将日历时间返回到 EAX 中
       add esp,4           ; 在调用之后清理堆栈
       mov [OldTime],eax ; 将时间值保存到内存变量中

; 使用 glibc 的 ctime() 函数产生一个本地时间的字符串概述:
       push OldTime        ; 将日历时间值的地址压入堆栈
       call ctime          ; 返回指向 EAX 中的 ASCII 时间字符串的指针
       add esp,4           ; 为 1 个参数做堆栈清理
       push eax            ; 将指向 ASCII 时间字符串的指针压入堆栈
       push TimeMsg        ; 将指向基础消息文本字符串的指针压入堆栈
       call printf         ; 合并且显示两个字符串
       add esp,8           ; 堆栈清理: 2 个参数*4 个字节 = 8

; 产生本地时间值到 glibc 的静态 tm 结构中:
       push dword OldTime; 将日历时间值的地址压入堆栈
       call localtime      ; 将指向静态时间结构的指针返回到 EAX 中
       add esp,4           ; 为 1 个参数做堆栈清理
;为 glibc 的静态 tm 结构做一份局部副本:
```

```
        mov esi,eax            ; 将静态 tm 的地址从 eax 中拷贝到 ESI 中
        mov edi,TmCopy         ; 将局部 tm 副本的地址放入 EDI 中
        mov ecx,9              ; 在 Linux 下，一个 tm 结构的大小是 9 个双字
        cld                    ; 将 DF 清零，以便我们可以进行上内存移动
        rep movsd              ; 将静态 tm 结构副本复制到局部副本中
; 显示 tm 结构中的一个字段:
        mov edx,dword [TmCopy+20] ; 年份字段在 tm 中的偏移地址是 20 个字节
        add edx,1900           ; 年份字段是自从 1900 年以来的年数
        push edx               ; 将值压入堆栈
        push YrMsg             ; 将基字符串的地址压入堆栈
        call printf            ; 用 printf 显示字符串和年份值
        add esp,8              ; 堆栈清理: 2 个参数*4 个字节 = 8
; 等几秒钟，以便让用户输入回车键，所以我们有了一个时间差:
        call getchar           ; 等待用户按 Enter 键
; 计算自程序开始运行以来已经过去的秒数:
        push dword 0           ; 将一个空指针压入堆栈;我们将会取出 EAX 中的值
        call time                       ; 获取当前的时间值: 返回到 EAX 中
        add esp,4                       ; 清理堆栈
        mov [NewTime],eax               ; 保存新的时间值
        sub eax,[OldTime]               ; 计算时间差的值
        mov [TimeDiff],eax              ; 保存时间差的值
        push dword [TimeDiff]           ; 将以秒计算的时间差的值压入堆栈中
        push Elapsed           ; 将已经流逝的时间的消息字符串的地址压入堆栈
        call printf            ; 显示已经流逝的时间
        add esp,8              ; 为 1 个参数做堆栈清理工作
;;; 在此之后的所有内容都是样板代码; 请在所有的一般应用程序中使用它!
        pop edi                ; 恢复已经保存的寄存器
        pop esi
        pop ebx
        mov esp,ebp            ; 在返回之前销毁堆栈帧
        pop ebp
        ret                    ; 将控制返回给 Linux
```

如果你到了 GNU 之外其他 Unix 实现中，一定要记住，time_t 值可能已经有了一些不同于 32 位整数的其他定义。这里，glibc 将 time_t 定义为一个 32 位整数，我们可以简单地通过将两个 time_t 值相减来计算它们之间时间差——至少一直到 2038 年。对于其他的非 GNU 实现的 Unix，最好使用 C 库函数 difftime()来返回两个 time_t 值之间的差值。

12.6 理解 AT&T 指令助记符

x86 CPU 拥有不止一个指令助记符集合，这已经成为很多混乱的根源。指令助记符只是一种为了让人类记住二进制位模式(如 1000100111000011)对于 CPU 的含义的手段。我们不是将 16 个 1 和 0 写在一行(甚至稍微容易记住的十六进制等值 89C3h)，而是将其说成“MOV BX,AX”。

请记住，助记符只是为人类设计的记忆唤起器——对于 CPU 本身而言是未知事物。汇编编译器将助记符翻译成机器指令。虽然我们可以商定将“MOV BX,AX”翻译成 1000100111000011，但是字符串“MOV BX,AX”并没有什么神秘的魔力。我们照样也可以商定用“COPY AX TO BX”或者“STICK GPREGA INTO GPREGB”来表示那个二进制位模式。我们之所以使用“MOV BX,AX”，是因为这是 Intel 的建议，由于 Intel 设计制造了 CPU 芯片，所以它可能最清楚如何描述自己产品的内部情况。

x86 指令助记符的候补者叫 AT&T 助记符，它产生于使 Unix 尽可能容易地移植到不同机器架构这一愿望。然而，操作系统实现者的目标与汇编语言程序员的目标不尽相同；如果目标是拥有一个完整的、最佳效率的 x86 处理器命令，使用英特尔系列编写代码会更好些，因为在整本书中我一直在讲授相关的内容。而如果目标在于理解 gcc 编译器和标准 C 库方面，则最好选择 AT&T，因为那是它的强项。

12.6.1 AT&T 助记符公约

gcc 将 C 源代码文件编译成机器代码时，它真正做的事情是将 C 源代码使用 AT&T 助记符翻译成汇编语言源代码。回去看一下图 12-1。gcc 编译器将 a.c 源代码文件作为输入，并输出一个以.s 为扩展名的汇编源代码文件，然后该文件被提交给 GNU 汇编编译器 gas 进行汇编。这就是 GNU 工具在所有平台上的工作原理。从某种意义上来讲，汇编语言是一种用于获取 C 编译器益处的中间语言。在大多数情况下，程序员根本不看它，也无须受它的愚弄。

这只是大多数情况。不过，如果打算处理标准 C 库和众多其他的用 C 语言为 C 而编写的库函数，至少熟悉一下 AT&T 助记符会很有意义。这里有一些一般的规则，一旦掌握了它们，熟悉过程将会变得更加容易。下面是简要描述列表：

- **AT&T 助记符和寄存器名称总是采用小写形式。**这是为了与 Unix 的大

小写习惯保持一致。我已经在正文和示例中把大写和小写混合在一起，目的是让你习惯于以这两种方式阅读汇编源文件，但是必须要记住，尽管 Intel(因此，NASM)建议使用大写，但是也会接受小写，而 AT&T 则要求必须小写。

- **寄存器名称总是以百分比符号%开头。**也就是说，Intel 写成 AX 或 EBX 的内容，AT&T 将写成%ax 和%ebx。这有助于汇编编译器识别出寄存器的名称。每一个具有操作数的 AT&T 机器指令助记符都有一个表明其操作数尺寸的单字符后缀。后缀字母有 b，w 和 l，表明字节(8 位)，字(16 位)，或更长的(32 位)。Intel 写成“MOV BX,AX”的内容，AT&T 会写成“movw %ax，%bx”。(%ax 和%bx 顺序发生改变并不是编写错误。见下一条规则！)
- **在 AT&T 语法中，源操作数和目标操作数存放的顺序刚好与英特尔语法相反。**也就是说，Intel 写成“MOV BX,AX”的内容，AT&T 公司将写成“movw %ax，%bx”。即：在 AT&T 语法中，源操作数在前面，后面的才是目标操作数。
- **在 AT&T 语法中，立即数操作数前面总是带有一个美元符号($)。**Intel 写成 PUSH 32 的内容，AT&T 将会写成 pushl $ 32。这有助于汇编编译器识别出立即数操作数。
- **并非所有 AT&T 指令助记符都由 gcc 产生。**与 Intel 的 JCXZ、JECXZ、LOOP、LOOPZ、LOOPE、LOOPNZ 和 LOOPNE 等值的相关指令只是最近才被添加到 AT&T 助记符集合中，gcc 从来不产生需要用到它们的代码。
- **在 AT&T 语法中，内存引用中的位移引用是有符号量，放在包含基址、索引和缩放比例值的括号外面。**稍后将单独讲解这一条，与之相关的内容将在.s 文件中多次出现，相信大家应该能够理解它。

12.6.2 查看 gcc 创建的 gas 源文件

要想大体了解 AT&T 汇编语法，最佳方式是观察一个由 gcc 产生的、实际具有 AT&T 风格的.s 文件。这样做有两个好处：首先，帮助我们熟悉 AT&T 的助记符和格式化公约。其次，在努力搞清楚如何从汇编中调用一个 C 库函数时，我们可能会发现，创建一个非常简短的、只调用我们感兴趣的函数的 C 程序，然后查看 gcc 编译程序时产生的.s 文件，这样做非常奏效。下面展示的 dateis.c 程序

实际上是我的 90 年代早期研究中的一部分，我当时用它来大概了解一下如何在汇编级别上调用 ctime( )。

每当编译一个 C 程序时，我们都不能自动获得.s 文件。.s 文件的确被创建了，但是一旦 gas 将.s 文件编译为二进制目标代码文件(通常是一个.o 文件)，它会删除该.s 文件。如果想查看一个由 gcc.s 创建的.s 文件，我们必须使用-S 选项编译。(注意，这是一个大写的 S！Unix 世界中的大小写非常重要！)该命令如下所示：

```
gcc dateis.c -S
```

请注意，此命令只是输出一个汇编源文件。如果指定了-S 选项，gcc 会了解到，我们希望它生成一个汇编源文件而不是一个可执行程序文件，因此它只生成.s 文件。要想将一个 C 程序编译成可执行程序文件，我们必须使用不带- S 选项的格式再次编译它。

这里是程序 dateis.c，它执行的任务只有一个：打印出标准 C 库函数 ctime()返回的日期和时间：

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
time_t timeval;
(void)time(&timeval);
printf("The date is: %s", ctime(&timeval));
exit(0);
}
```

这不是一个很了不起的程序，但是它确实演示了三个 C 库函数调用的使用：time()、ctime()和 printf()。当 gcc 编译前面的程序(dateis.c)时，将产生如清单 12-4 所示的 dateis.s 文件。我已经在 AT&T 助记符的右侧，手动添加了与之等值的英特尔助记符作为注释，所以可以清楚地看到在两个系统中哪些命令等同于哪些命令。(唉，可惜无论是 gcc 还是任何我见过的其他实用工具都不能自动完成这件事情。)

清单 12-4：dateis.s

```
    .file           "dateis.c"
    .version        "01.01"
```

```
    gcc2_compiled.:
    .section .rodata
    .LC0:
        .string "The date is: %s"
    .text
        .align 4
    .globl main
        .type main,@function
    main:
        pushl %ebp              # push ebp
        movl %esp,%ebp          # mov ebp,esp
    subl $4,%esp                # sub esp,4
        leal -4(%ebp),%eax      # lea eax,ebp-4
        pushl %eax              # push eax
        call time               # call time
        addl $4,%esp            # add esp,4
        leal -4(%ebp),%eax      # lea eax,ebp-4
        pushl %eax              # push eax
        call ctime              # call ctime
        addl $4,%esp            # add esp,4
        movl %eax,%eax          # mov eax,eax
        pushl %eax              # push eax
        pushl $.LC0             # push dword .LC0
        call printf             # call printf
        addl $8,%esp            # add esp,8
        pushl $0                # push dword 0
        call exit               # call exit
        addl $4,%esp            # add esp,4
        .p2align 4,,7
    .L1:
        leave                   # leave
        ret                     # ret
    .Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
```

当阅读这段代码时，需要记住一点：dateis.s 包含的是由编译器机械产生的汇编语言代码，不是由人类程序员编写的代码！关于这些代码，有些内容(如：为什么标签.L1 存在但却从未被引用过)可能不够理想，我们只能将其解释为 gcc 编译机制的“加工品”。在某些更为复杂的程序中，可能会有一些在此类简单程序

中用不到的关于标签.L1 的习惯用法。

在阅读前面的程序清单时，这里有一些需要注意的小事项：

- 当指令不采用操作数(call, leave, ret)时，它不需要操作数大小后缀。调用和返回在 Intel 中和在 AT&T 中的语法非常相像。
- 当被引用时，消息字符串的名称带有一个美元标志($)前缀，就像立即数一样。而在 NASM 中，一个已命名的字符串变量则被认为是一个变量而不是立即值。这是另外一个需要注意的 AT&T 小瑕疵。
- 注意，AT&T 架构中的注释界定符是英镑符号(#)，而不是在几乎所有的 Intel 式汇编程序(包括 NASM)中使用的分号。

12.6.3 AT&T 内存引用语法

正如我们可能会从前述章节中记起的那样，引用一个内存位置(这与引用它的地址不同)可以通过将该位置放在方括号内部来实现，例如：

```
mov ax, dword [ebp]
```

这里，我们正在谈论任意一个位于 EBP 中所包含的地址处的 32 位量，并且将它加载到寄存器 AX 中。更为复杂的内存寻址可以是这样的：

```
mov eax, dword [ebx-4]        ; 基址减去位移量
mov al, byte [bx+di+28]       ; 基址加上索引加上位移量
mov al, byte [bx+di*4]        ; 基址加上索引乘以缩放比例
```

上述示例全部采用 Intel 语法。AT&T 用于内存寻址的语法有很大不同。它没有采用方括号而是使用小括号来圈住内存地址部分：

```
movb (%ebx),%al               # mov byte al,[ebx] 在 Intel 语法中
```

这里，我们正在将位于[ebx] 中的字节量移动到 AL 中。(不要忘记，操作数的顺序与 Intel 的语法正好相反。)如果存在基址、索引和缩放比例相关内容，应该将它们放在小括号内。(基址必须总是存在的。)如果有一个位移的话，该位移必须放在小括号的前面，并且放在小括号的外面：

```
movl -4(%ebx),%eax            # mov dword eax,[ebx-4] 在 Intel 语法中
movb 28(%ebx,%edi),%eax        # mov byte eax,[ebx+edi+28]在 Intel 语
法中
```

请注意，在 AT&T 语法中，括号内的内容并不采用算术运算的形式表达。基址、索引和缩放比例以逗号进行分隔，并且不允许使用加号和星号。用于解释 AT&T 内存引用的架构如下：

```
±disp(base,index,scale)
```

上述演示示例中使用的±符号表明，位移是有符号的；也就是说，它可以为正也可以为负，指明该位移值是与其余的地址相加还是相减。通常情况下，我们只看到显式的负号；如果没有减号符号，则假定位移为正。位移量和缩放比例值都是可选项。

但是，大部分时间里我们看到的将是非常简单的内存引用形式：

```
-16(%ebp)
```

当然，位移会有所不同，但是它几乎总是意味着这样的内容：指令正在引用一个位于堆栈某处的数据项。C 代码将它的变量分配到堆栈上，在一个堆栈帧中，然后通过来自 EBP 的值所代表的字面偏移量引用那些变量。EBP 相当于一个“堆栈中的标杆”，堆栈上的数据项可以通过相对于 EBP 的偏移量(或者为正，或者为负)来进行引用。前面的引用告诉一条机器指令处理一个位于 EBP 中的地址减去 16 个字节后的位置处的数据项。

12.7　产生随机数

作为再次步入从汇编语言调用标准 C 库的快速旅行，让我们变得更加随机些。(或者至少要达到伪随机。)标准 C 库具有一对儿函数能够让程序产生伪随机数。这里，伪(*pseudo*)这个前缀是非常重要的。研究表明，没有任何可证明的方式能够严格从软件产生真正的随机性以及随机数。事实上，随机所代表的整个概念有些神经质，这些天已经难住了很多数学家。从理论上讲，要想实现真正的随机，你的确需要从某种量子现象中获得一些触发机制(放射现象是最经常提及的)。但是，由于缺乏一个核动力随机数产生器，我们将不得不退而求其次，接受“伪”随机。

可以采用类似这样的内容来简化伪随机数的定义：一个由伪随机数产生器产生没有可识别模式的数字序列，但是这个序列可以通过向该产生器传递相同的随机种子而重复生成。种子值只是一个输入到某个创造伪随机数序列的神秘算法中的整数值。通过向产生器传递相同的种子，我们可以得到相同的序列。然而，在

这个序列内部，发生器范围之内的数字分布得相当分散和随意。

标准 C 库包含两个与伪随机数相关的函数：

- srand()函数传递一个新的种子值给随机数产生器。该值必须是一个 32 位整数。如果没有传递种子值，则该值默认为 1。
- rand()函数返回一个 31 位的伪随机数。最高位一直为 0，因此，如果作为 32 位有符号整数处理的话，该值始终为正。

一旦了解这些函数的工作原理，使用它们就非常容易了。

12.7.1　利用 srand()为随机生成器“播种”

实际上，为生成器传入种子值比进行将下一个伪随机数拉入当前序列的调用更加难缠。这并不是说 srand()调用也那么困难：将种子值压入堆栈，然后就可以调用 srand()了：

```
push eax            ; 这里，种子值存放在 eax 中
call srand          ; 向伪随机数产生器传递该种子值
add esp,4           ; 为一个参数做堆栈清理工作
```

这就是必须要做的所有事情！srand()函数不返回任何值。但是……使用什么作为种子的值呢？

唉，这就是困难所在。

如果对于程序而言，每次运行时都不使用精确相同的伪随机数序列非常重要，我们显然不希望使用一个普通的整数硬编码到程序中。在每次运行该程序时，理想的情况是要得到不同的种子值。做这件事情的最佳方法(虽然还有其他方法)是在调用 srand()时，向其传递自 1970 年 1 月 1 日以来的秒数作为种子，该值可通过 time()函数返回，这已经在上一节中进行了讲解。这个名为 time_t 的值是一个有符号整数，它每秒钟变化一次，所以，每过去一秒，都可以有一个新的种子值，而且是一个从定义上来讲永远不会重复的值。(在此我们假设上一节中提到的 time_t 翻转问题将会在 2038 年以前得到解决)。

几乎每个人都是这样做的，唯一需要小心的是，必须保证不要过于频繁地调用 srand()来重新“播种”序列，以至于每秒调用不止一次。大多数情况下，对于即将运行的程序而言，执行其工作以及执行完毕后结束其工作只需几分钟或者几小时的时间，这样，我们只需在程序开始执行时调用 srand()一次就可以了。如果正在编写一个将持续运行数天或数周或更长时间而一直不终止的程序，(比如：

服务器)，建议每天都为随机数产生器赋予一个新的种子。

这里有一个简短的代码序列，它调用 time()来检索当前的 time_t 值，然后将这个值提交给 srand()：

```
push 0          ;将一个 32 位空指针压入堆栈
call time       ;返回 time_t 值(一个 32 位整数)到 eax 中
add esp,4       ;清理堆栈
push eax        ;将 eax 中的时间值压入堆栈中
call srand      ;时间值就是随机生成器的种子.
add esp,4       ;清理堆栈
```

最开始的空指针压栈操作告诉 time()，并没有向它传入一个变量以接收这个时间值。但是，那个空指针必须在那里，这样 time()函数才能高高兴兴地工作。想要保存的值被返回到 EAX 中。

12.7.2 产生伪随机数

一旦为产生器传递了种子值，获取伪随机数序列中的数字十分简单：每次调用 rand()时都可将下一个数字拉入该序列。像 C 库中的所有其他函数一样，rand()函数非常容易使用：它没有参数(所以不需将任何内容压入堆栈或者事后进行清理堆栈)，并且伪随机数被返回到 EAX 中。

清单 12-5 中的程序 randtest.asm 演示了 srand()和 rand()的工作原理。它还展示了几个有趣的汇编技巧，我将在本节的其余部分对它们进行讨论。

清单 12-5：randtest.asm

```
; 源文件名字      : RANDTEST.ASM
; 可执行程序名    : RANDTEST
; 版本            : 2.0
; 创建日期        : 12/1/1999
; 最后修改日期    : 5/22/2009
; 作者            : Jeff Duntemann
; 描述            :一个 Unix rand 和 srand 的演示程序，使用 NASM 2.05
;
; 使用以下命令生成：
; nasm -f elf -g -F stabs randtest.asm
; gcc randtest.o -o randtest

[SECTION .data]              ; 包含已初始化的数据的段
```

```
    Pulls dd 36                  ; 我们打算拉入多少个数字?
    Display db 10,'Here is an array of %d %d-bit random
numbners:',10,0
    ShowArray db '%10d %10d %10d %10d %10d %10d',10,0
    CharTbl db
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz-@'

    [SECTION .bss]               ; 包含未初始化数据的段

    BUFSIZE equ 70               ; 随机选择字符的个数
    RandVal resd 1               ; 保留一个整数变量
    Stash resd 72                ; 为随机数保留一个由 72 个整数组成的数组
    RandChar resb BUFSIZE+5      ; 用于存储随机选择字符的缓冲区

    [SECTION .text]              ; 包含代码的段

    extern printf
    extern puts
    extern rand
    extern scanf
    extern srand
    extern time

    ;-------------------------------------------------------------------
    ; 随机数生成器过程—最后修改日期 5/22/2009
    ; 此例程提供了 5 个入口点，并返回 5 种不同“大小”的基于 rand 返回值的伪随机数。
    ; 注意：首先，rand 拉入的是一个 31 位的值。高 16 位是最"随机”的，
    ; 所以为了返回一个较小范围内的数字，可以取出一个 31 位的值，然后将它右移，
    ; 除了想要的那些位之外的其余位均以零补齐。一个 8 位的随机值范围是从 0-255，
    ; 一个 7 位的随机值范围是 0~127，等等。
    ; 尊重 EBP、 ESI、 EDI、 EBX 和 ESP。将随机值返回到 EAX 中.
    ;-------------------------------------------------------------------
    pull31: mov ecx,0            ;对于 31 位随机数，不需移位
        jmp pull
    pull16: mov ecx,15           ;对于 16 位随机数，移动 15 位
        jmp pull
    pull8: mov ecx,23            ; 对于 8 位随机数，移动 23 位
        jmp pull
    pull7: mov ecx,24            ; 对于 7 位随机数，移动 24 位
        jmp pull
    pull6: mov ecx,25            ; 对于 6 位随机数，移动 25 位
        jmp pull
```

```
pull4: mov ecx,27             ; 对于 4 位随机数，移动 27 位
pull: push ecx                ; rand 能够破坏 ecx 的值;所以将移位值保存到堆栈上
    call rand                 ; 调用 rand 来获取随机值;并将其返回到 EAX 中
    pop ecx                   ; 将藏起来的移位值弹回到 ECX 中
    shr eax,cl                ; 根据选择的因子对随机值进行移位
                              ; 记住，我们想要的部分是在 CL 中
    ret                       ; 以将随机数放入 EAX 中而返回;

;----------------------------------------------------------------------
; 创建新行 outputter - 最后更新日期 5/22/2009
; 此例程可以输出很多个换行符到标准输出，换行符的个数由传递到 EAX 中的值来决定。
; 合法的值是 1-10。所有神圣的寄存器都得到“尊重”。
; 如果传递 0 到 EAX 中，将导致没有换行符发出。
;----------------------------------------------------------------------
newline:
    mov ecx,10                ; 我们需要一个跳过的值，它是
    sub ecx,eax               ; 10 减去主调程序想要调用的换行符的个数
    add ecx,nl                ; 在调用 printf 之前，这个跳过的值被与换行符
; 缓冲区 nl 的地址相加
    push ecx
    call printf               ; 显示选定数量的换行符
    add esp,4                 ; 为一个参数做堆栈清理工作
    ret                       ; 返回
nl db 10,10,10,10,10,10,10,10,10,10,0

;; 这个子程序能够显示数字，一次显示 6 个
;; 不是旨在一般的用途......
shownums:
    mov esi, dword [Pulls]        ; 将拉入的随机数的计数值放入 ESI 中
.dorow: mov edi,6                 ; 将行元素计数器放入 EDI 中
.pushr: dec edi                   ; 递减行元素计数器
    dec esi                       ; 递减拉入的随机数的计数器的值
    push dword [Stash+esi*4]      ; 将来自数组的数字压入堆栈中
    cmp edi,0                     ; 我们已经将这行填满了吗?
    jne .pushr                    ; 如果没有,继续压入另一个随机值
    push ShowArray                ; 将基础显示字符串的地址压入堆栈
    call printf                   ; 显示随机数
    add esp,28                    ; 堆栈清理: 7 项 X 4 个字节 = 28
    cmp esi,0                     ; 判断一下拉入随机数的计数值是否已经为 0
    jnz .dorow                    ; 如果没有，我们回去继续生成另一行!
    ret                           ; 任务完成,所以返回!
```

```
;; 这个子程序拉出一个随机值，并将它们插入一个整数数组。不是旨在一般用途。
;; 请注意，在这个程序被调用之前，随机数产生器的入口点地址必须被加载到 edi 中，
;; 否则会引发错误.
puller:
    mov esi,dword [Pulls]          ; 将拉入的随机数的数量放到 ESI 中
.grab: dec esi                     ; 递减 ESI 中的计数器值
    call edi ; Pull the value      ; “拉出”一个值并将其返回到 eax 中
    mov [Stash+esi*4],eax          ; 将存储随机值存储到数组中
    cmp esi,0                    ; 判断一下是否已经拉入了足够数量的随机数
    jne .grab                    ; 如果 ESI <> 0 ，继续拉入另外一个
    ret                          ; 否则，返回！
                                 ; 主程序

global main                      ; 这条指令必须有，以便连接器能够找到入口点

main:
    push ebp                     ; 为调试器建立堆栈帧
    mov ebp,esp
    push ebx                     ; 程序必须保存 EBP、EBX、ESI 和 EDI 的值
    push esi
    push edi
;;; 在此之前的所有内容都是样板代码；请在所有的一般应用程序中使用它！

;以赋予随机数产生器一个 time_t 值作为种子作为开始：
Seedit: push 0               ; 将一个 32 位空指针压入堆栈
    call time                ; 返回 time_t 值(32 位整数)到 EAX 中
    add esp,4                ; 为一个参数做堆栈清理工作
    push eax                 ; 将 EAX 中的 time_t 值压入堆栈
    call srand               ; Time_t 值是随机数产生器的种子值
    add esp,4                ; 为一个参数做堆栈清理工作

; 除了正在生成的随机值的尺寸不同之外，下面的代码块基本相同：

; 创建和显示一个 31 位随机值的数组
    mov edi,pull31           ; 将#随机子函数的地址复制到 EDI 中
    call puller              ; 根据[pulls]中的值的需要，拉入指定数量的数字
    push 32                  ; 被拉入的数字的尺寸,以位为单位
    push dword [Pulls]       ; 产生的随机数的数量
    push Display             ; 基础显示字符串的地址
    call printf              ; 显示标签
    add esp,12               ; 堆栈清理： 3 个参数*4 个字节 = 12
    call shownums            ; 显示随机数行
```

```
; 创建和显示一个 16 位随机值数组
    mov edi,pull16          ; 将#随机子函数的地址复制到 EDI 中
    call puller             ; 根据[pulls]中的值的需要，拉入指定数量的数字
    push 16                 ; 被拉入的数字的尺寸,以位为单位
    push dword [Pulls]      ; 产生的随机数的数量
    push Display            ; 基础显示字符串的地址
    call printf             ; 显示标签
    add esp,12              ; 堆栈清理: 3 个参数*4 个字节 = 12
    call shownums           ; 显示随机数行

; 创建和显示一个 8 位随机值数组:
    mov edi,pull8           ; 将#随机子函数的地址复制到 EDI 中
    call puller             ; 根据[pulls]中的值的需要，拉入指定数量的数字
                            [pulls]中
    push 8                  ; 被拉入的数字的尺寸,以位为单位
    push dword [Pulls]      ; 产生的随机数的数量
    push Display            ; 基础显示字符串的地址
    call printf             ; 显示标签
    add esp,12              ; 堆栈清理: 3 个参数*4 个字节 = 12
    call shownums           ; 显示随机数行

; 创建和显示一个 7 位随机值数组:
    mov edi,pull7           ; 将#随机子函数的地址复制到 EDI 中
    call puller             ; 根据[pulls]中的值的需要，拉入指定数量的数字
    push 7                  ; 被拉入的数字的尺寸,以位为单位
    push dword [Pulls]      ; 产生的随机数的数量
    push Display            ; 基础显示字符串的地址
    call printf             ; 显示标签
    add esp,12              ; 堆栈清理: 3 个参数*4 个字节 = 12
    call shownums           ; 显示随机数行

; 创建和显示一个 4 位随机值数组:
    mov edi,pull4           ; 将#随机子函数的地址复制到 EDI 中
    call puller             ; 根据[pulls]中的值的需要，拉入指定数量的数字
    push 4                  ; 被拉入的数字的尺寸,以位为单位
    push dword [Pulls]      ; 产生的随机数的数量
    push Display            ; 基础显示字符串的地址
    call printf             ; 显示标签
    add esp,12              ; 堆栈清理: 3 个参数*4 个字节 = 12
    call shownums           ; 显示随机数行
```

```
; 将一个缓冲区清空:
    Bufclr: mov ecx, BUFSIZE+5    ; 为了安全起见，填充整个缓冲区+5
    .loop: dec ecx                ; BUFSIZE 是基于 1 的，所以先递减!
           mov byte [RandChar+ecx],0 ; 将空值移入缓冲区中
           cmp ecx,0               ; 一切都结束了吗?
           jnz .loop               ; 如果没有，回去继续填充下一个空值

; 创建一个由随机字母数字字符构成的字符串:
    Pulchr: mov ebx, BUFSIZE      ; BUFSIZE 告诉我们拉入多少个字符
    .loop: dec ebx                ; BUFSIZE 是基于 1 的，所以先递减!
           mov edi,pull6          ; 对于范围在 0 到 63 之间的随机值
           call puller            ; 获取一个范围在 0 到 63 之间的随机值
           mov cl,[CharTbl+eax]   ; 使用 eax 中的随机数 #作为表内的偏移量，
; 并且将字符从表中复制到 CL
           mov [RandChar+ebx],cl  ; 将字符从 CL 中复制到字符缓冲区
           cmp ebx,0              ; 在获得乐趣的同时完成任务了吗?
           jne .loop              ; 如果还没有，回去继续拉入另一个随机值
; 显示随机字符构成的字符串:
           mov eax,1              ; 输出一个换行符
           call newline           ; 使用 newline 过程
           push RandChar          ; 将字符缓冲区的地址压入堆栈
           call puts              ; 调用 puts 将其显示出来
           add esp,4              ; 为一个参数做堆栈清理工作
           mov eax,1              ; 输出一个换行符
           call newline           ; 使用 newline 子程序
;;;至此以后的所有内容都是样板，请在所有的一般应用程序中使用它!
           pop edi                ; 恢复保存的寄存器
           pop esi
           pop ebx
           mov esp,ebp            ; 在返回之前，销毁堆栈帧
           pop ebp
           ret                    ; 将控制返回给 Linux
```

12.7.3 有些位比其他位更具随机性

在 Linux 下，rand()函数以一个 32 位整数来返回一个 31 位无符号值。(该整数的标志位——所有 32 位的最高位——总是被清除为 0。)有关 rand()和 srand()的 Unix 文档表明，由 rand()产生的值的低序位总是不如高序位随机性强。这意味着，如果只打算使用由 rand()产生的部分位，应该尽量使用那些最高序的位。

我真的不知道为什么会这样，也不知道这个问题有多么糟糕。我不是一个研

究数学的人，我愿意相信那些 rand()文档的编写者所说的话；但是它与如何限制打算产生的随机数的范围这一问题有关。

这个问题是显而易见的：假设想拉入 ASCII 字符中的一个随机字母的数字代码。我们不需要从 0～20 亿之间的所有数字。目前只有 127 个 ASCII 字符，并且实际上只有 62 个是字母和数字。(其余的为标点符号、空格、控制字符或不可打印字符，如笑脸符。)我们想要做的就是拉出某些 0 到 61 之间的随机数。

从 0～20 亿范围内拉出一些数字，直到找到一个小于 62 的值，这将需要很长的时间。显然，我们需要一种其他的方法。我采用的方法是将 rand()返回的 31 位随机值作为一个随机位的集合。然后从这些位中摘录一个刚好满足需要的子集。6 位可以表达从 0～63 之间的值，所以我从原来的 31 位值中拿出最高的六个位，并使用它们来指定随机字符。

这很容易实现：只需将该 31 位值右移，直到除了最高的 6 个位之外的所有其他位都被从该值的右侧移出。同样的技巧适用于任何(合理)数量的位。需要做的所有事情就是选择要移动的位数。我已经创建了一个具有多个入口的程序，每个入口点从随机值中选出一定数量的位数：

```
pull31: mov ecx,0       ; 对于 31 位随机数,我们不移位
        jmp pull
pull16: mov ecx,15      ; 对于 16 位随机数,我们移动 15 位
        jmp pull
pull8: mov ecx,23       ; 对于 8 位随机数,我们移动 23 位
        jmp pull
pull7: mov ecx,24       ; 对于 7 位随机数,我们移动 24 位
        jmp pull
pull6: mov ecx,25       ; 对于 6 位随机数,我们移动 25 位
        jmp pull
pull4: mov ecx,27       ; 对于 4 位随机数,我们移动 27 位
pull: push ecx          ; rand() 使 ECX 变成了垃圾;所以将移位值保存到堆栈上
        call rand       ; 为了获取随机值调用 rand(); 返回值存放到 EAX 中
        pop ecx         ; 将保存的移位值弹出到 ECX 中
        shr eax,cl      ; 根据选择的因子将随机值移动相应的位数
                        ; 记住，我们想要的那部分是在 CL 中
        ret             ; 将随机数存放到 EAX 中，返回
```

要想拉出一个 16 位的随机数，调用 pull16。要想拉出一个 8 位的随机数，调用 pull8，等等。我的确发现，较小数字的随机性不如较大数字的强，并且由 pull4 产生的数字可能不会随机到足够有用的程度。(我暂时搁置 pull4 代码，所以

大家可以自己通过运行 randtest 看一下。)

这里的逻辑应该很容易理解：首先选择一个移位值，将其放入 ECX 中，然后将 ECX 压入堆栈，调用 rand()，再从堆栈中弹出 ECX，然后将(rand()返回在 EAX 中的)随机数移动 CL 中的值所指定的位数——当然，CL 是 ECX 寄存器的最低的 8 位。

为什么要将 ECX 放到堆栈上呢？在 C 调用约定中，ECX 并不是神圣的寄存器之一，并且几乎所有的 C 库例程都在内部使用 ECX，因此会破坏它的值。如果想在一次库函数调用过程中保留 ECX 的值，必须确保在调用之前将该值保存在某些地方，然后在调用完成后再将其恢复。堆栈是做这件事情的最佳场所。

我使用 pull6 例程来拉出一个 6 位的随机数字，以从字符表中选择一个字符，从而建立一个随机字母数字字符的字符串。我用 64 个元素来填充这张表，并添加了两个额外的其他字符(- 和@元素)，这样我就不必测试每个拉入的数字，看看它是否小于 62 了。如果你需要将一个随机值限制在一定的范围内，而该范围不是 2 的幂数，则选择第二大的 2 的幂数——但是要尽量设计一下程序，使其不必选择在诸如 0～65 范围之内的随机值。很多关于随机数算法的书籍已经问世，因此，如果对这个概念比较着迷的话，我建议你去那些书中做进一步的研究。

12.7.4 调用寄存器中的地址

我在 randtest 中使用了一种有时会被汇编新手遗忘的技术：可以执行一条 CALL 指令来调用一个地址存放在寄存器中的过程。我们并不总是必须使用带有立即标签的 CALL 指令。换句话说，下面两个代码片段中的 CALL 指令是完全合法且等价的：

```
mov ebx, pull8      ; 将 pull8 标签代表的地址加载到 EBX 中
call pull8          ; 调用由 pull8 所代表的地址
call ebx            ; 调用存放在 EBX 中的地址
```

为什么要这样做呢？随着时间的推移，我们肯定会找到各自不同的理由，但是从总体上来讲，它能够让我们把过程调用当作参数来使用。在 randtest 中，我将很多代码分解到一个名为 puller 的过程中，然后几次调用 puller 来产生不同尺寸的随机数。我通过将那个过程的地址加载到 EDI 中来向 puller 传递正确的随机数过程的地址：

```
;创建和显示一个 8 位随机数数组：
mov edi,pull8           ; 将#随机子函数的地址复制到 edi 中
call puller             ; 根据[pulls]中的值的需要，拉入指定数量的数字
```

下述的 puller 过程采用这样的方式调用请求的随机数过程：

```
puller:
    mov esi,dword [pulls]     ; 将拉入的随机数的数量放入 ESI 中
.grab: dec esi                ; 递减存放在 ESI 中的计数值
    call edi                  ; 拉入这个值；并将其返回到 EAX 中
    mov [stash+esi*4],eax     ; 将随机值存储到数组中
    cmp esi,0                 ; 判断一下我们是否已经拉入了足够多的随机值
    jne .grab                 ; 如果 ESI<> 0，拉入另外一个随机值
    ret                       ; 否则，返回！
```

看到 CALL EDI 指令了吗？在这种情况下(EDI 已经事先加载了 pull8 过程的地址)，尽管标签“pull8”并没有在 puller 过程中出现，但是调用的却是 pull8 过程。通过将不同过程的地址传递到 EDI 中，我们可以使用 puller 中的同样的代码向缓冲区中填充不同大小的随机数。

我们可以调用存放在寄存器中的地址来一般化我们的代码——但是需要确保把正在做的事情记入文档中，因为正在调用的标签并没有包含在 CALL 指令中！

12.8 C 如何看待命令行参数

在第 11 章中，我解释了如何从一个 Linux 程序中访问命令行参数，并将其作为对堆栈帧的更广泛讨论中的一部分。在 glibc 中连接和调用标准 C 库中的函数时，有一件不可思议的事情，那就是访问命令行参数的方式会有所变化，而且变化显著。

参数仍然在堆栈中，这就是参数地址表。(如果你没有通过第 11 章中的那一节，请现在就对它进行研究，否则本节将没有多大意义。)问题是 glibc 的启动代码也把一些其他内容放在那里，而这些其他内容会对我们有些妨碍。

幸运的是，有一种方法能够通过绕过这些障碍，(它们包括很多重要内容，比如：将执行带入 glibc 关闭代码以允许程序体面退出的返回地址)。办法就是通过“堆栈中的标杆”：EBP。EBP 锚定了对于自己的数据项(采用从 EBP 开始的内存往下存放)以及那些被 glibc 放入堆栈的、采用上内存方式存储的数据项的访问。

Glibc 为堆栈帧添加了一个指向那张指向参数本身的地址表的指针。因为我们谈论的指针指向一张元素为指向实际参数字符串的指针的表，所以最好的入手方法是绘制一张相关情况图。图 12-4 显示了要想识别和读出命令行参数所必须了解的指针关系和堆栈结构。

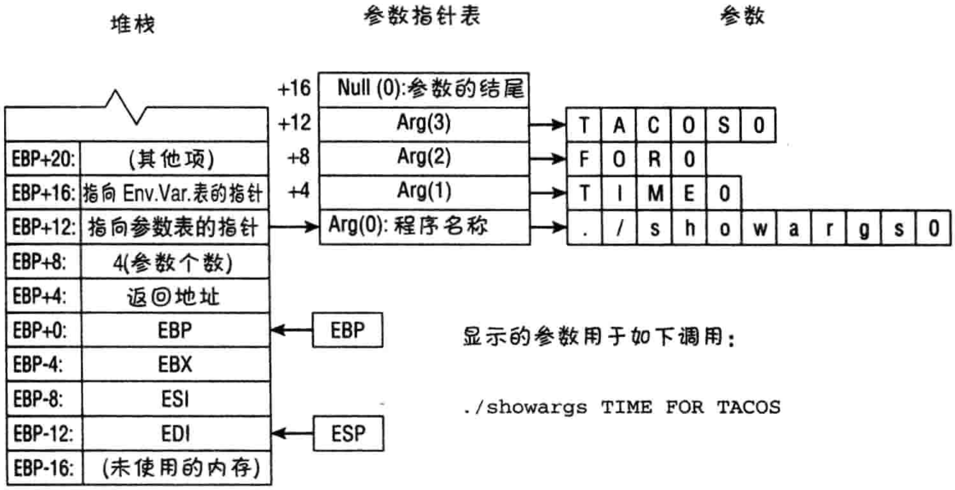


图 12-4　当存在 C 库时访问命令行参数

紧跟在 EBP 上面的是你那部分程序的返回地址。当代码执行完毕时，它执行一个 RET 指令，而 RET 正是使用这个返回地址将执行带回 C 库的关闭序列。我们不需为了任何事情而直接访问这个返回地址；当然，我们也不应该修改它。

在返回地址的紧上方，距离 EBP 偏移量为 8(有些文献也将其描述为 EBP+8)的地方是一个反映参数数量的整数计数值。不要搞混：这是一个参数数量的副本，像 11 章中所说的那样，它出现在参数地址表的下面。

仅在参数计数器上面，在 EBP+12 的位置，是一个指向参数地址表的指针。仅在它的上面，在 EBP+16 的位置，存放的是一个指向环境变量对儿表的指针。读取环境变量对儿的方式与读取命令行参数的方式大致相同，因此，如果已经理解了其中一个，理解另一个就不会有麻烦。

即使有很多额外的间接操作，显示参数列表所需的代码也少得出奇：

```
    mov edi,[ebp+8]          ; 将参数数量加载到 EDI 中
    mov ebx,[ebp+12]         ; 将指向参数表的指针加载到 EBX 中
    xor esi,esi              ; 将 ESI 清零
.showit:
```

```
    push dword [ebx+esi*4]      ; 将一个参数的地址压入堆栈
    push esi                    ; 将参数数量压入堆栈
    push ArgMsg                 ; 将显示字符串的地址压入堆栈
    call printf                 ; 显示实参编号和实参
    add esp,12                  ; 堆栈清理：3 个参数*4 个字节 = 12
    inc esi                     ; 从参数#跳到下一个参数
    dec edi                     ; 将参数计数器的值递减 1
    jnz .showit                 ; 如果参数计数器的值为零,我们的任务已经完成
```

参数计数值和参数表的地址都在相对于 EBP 固定偏移量的位置处。参数计数值存放在 EDI 中。参数表的地址存放在 EBX 寄存器中。ESI 被清零，并提供一个相对于参数地址表的偏移量。做完之后，我们进入一个循环，这个循环将参数指针、参数数量，以及一个基字符串压入堆栈并调用 printf()来显示它们。每当打印一个参数时，我们都递增存放在 ESI 中的参数数量，并递减存放在 EDI 中的参数计数值。当 EDI 变为 0 时，我们已经显示了所有的参数，大功告成！

关于这个程序的一个重要注意事项，我在前面已经说过，这里必须要再次强调一下：如果正在一个循环中调用某个 C 库函数，要么必须使用神圣的寄存器来保存管理循环的计数值，要么在进行 C 库调用之前，必须将其压入堆栈。该库能使不受保护的寄存器 EAX、ECX 和 EDX 中的值变为垃圾。如果试图将参数计数值存储到 ECX 中，那么，该计数值在第一次调用 printf()时就会遭到破坏了。EBX、ESI 和 EDI 的神圣的本质使得它们成为此类用法的理想选择。

在本书的清单列表中，有一个包含了前述代码的完整程序，它就是 showargs3.asm。

12.9 简单文件输入/输出

我在本书中提供的最后一个示例程序名义上是关于处理基于磁盘的文本文件的。但是，它也包含了很多前面讲过的汇编技巧，并且添加了更多的新技巧。这是本书中规模最为庞大，也最为复杂的程序，如果你能够读懂它，并能跟进逻辑的流动，说明你已经得到了我企图在本书中教给你的一切知识。它比本书中的其他任何内容更像一个“真正的”程序，因为它使用命令行参数工作，将输出写入一个磁盘文件，并且能够做一些创建任何实用工具都很可能需要的其他有用事情。

清单 12-6 中的 textfile.asm 程序创建并填充一个文本文件。我们可以指定即将填充到文件中的行数，以及这些行的文本。如果没有指定该文件的文本，这个

程序将产生一行随机选择的字符，并且使用它来取代文本。可以这样调用该程序：

```
$./textfile 150 Time for tacos!
```

上述调用将会创建一个新的文件(在程序中，这个新文件的名字固定为“testeroo.txt”)并且在关闭该文件之前，将文本“吃玉米饼的时间！”写入该文件中 150 次。如果文件 testeroo.txt 已经存在，它被从起始处开始覆盖。如果没有在行计数值后面键入任何内容，该程序将采用随机字母数字字符填充该文件。如果没有键入一个整数作为第一个参数，textfile 将显示错误消息。如果只输入程序名称，然后按下 Enter 键，textfile 将解释它的功能并显示有关其使用方法的几行信息。

12.9.1　通过 sscanf()将字符串转换为数字

当调用一个程序时，如果在命令行中输入了一个数字，我们能够通过本章前面所描述的机制将其作为一个命令行参数来进行访问。然而，这里有一个问题：因为数字是作为文本呈现的，例如，不能只采用文字字符串"751"并将其加载到一个寄存器或一个整数变量中。要想将数字命令行参数作为数字来使用，必须将它们从文本表达形式转换成数字形式。

标准 C 库中有几个函数能够应对这一挑战。其中一些(比如 strtod)非常具体而且功能有限，只能将文本转换为一种数值类型。然而，有一个 C 函数几乎能够将任何一个合法数字值的文本表达转换成一个适当的数字表达形式。它就是 sscanf()，也是我们将在清单 12-6 中使用的那个。

sscanf()函数需要三个参数，我们必须按照以下顺序将它们压入堆栈：

(1) 首先，将包含 sscanf()所产生的数字值的数字型变量的地址压入堆栈。这里，我们生成了一个 32 位整数，所以，在 textfile.asm 中我们传递内存变量 LineCount 的地址，它是一个 32 位整数。

(2) 现在，将格式化代码字符串的地址压入堆栈，它告诉 sscanf()我们想把输入的文字转换成哪种数字格式。这里的代码字符串是“%d”，可从关于 printf()的讨论中回想关内容，它是专门处理双字(32 位整数)的代码。

(3) 最后，将即将被转换成其代表的数字值的文本字符串的地址压入堆栈。在 textfile.asm 中，我们将 arg(1)的地址压入堆栈，这是当调用程序时，在命令行中键入的第一个命令行参数。

一旦将这三个地址压入堆栈之后，调用 sscanf()。它将转换后的值返回到已将其地址作为第一个参数传递的数字变量中。它还返回一个编码到 EAX 中，以指示转换是否成功。如果 EAX 中的返回值为 0，表明发生了一个错误，此时你不要指望在数字变量中有任何有用的数据。如果转换成功，我们将会在 EAX 中看到值 1。

这是使用 sscanf()的最简单的方式。它也能够一次转换整个数字数组，但那是一种更加专业的用法，在刚刚开始学习汇编的时候未必需要。将其地址作为第三个参数传递给 sscanf()的字符串应该包含格式描述字符串中每一个格式化代码所代表的数字值的文本。

整个过程如下所示：

```
mov ebx,[ebp+12]   ; 将指向参数表的指针放入 ebx 中
push LineCount     ; 为了调用 sscanf，将代表行数的整数的地址压入堆栈。
push IntFormat     ; 将整数格式化代码的地址压入堆栈
push dword [ebx+4]; 将 arg(1)的地址压入堆栈
call sscanf        ; 调用 sscanf 将 arg(1)转换成一个整数
add esp,12         ; 堆栈清理： 3 个参数*4 个字节 = 12
cmp eax,1          ; 返回值 1 说明我们得到了一个有效的数字
je chkdata         ; 如果我们得到了一个有效的数字，继续；否则异常终止
```

假设用户至少在命令行中输入一个参数(并且此程序已经验证了这一点)，则指向第一个参数的指针位于相对于命令行参数指针表的开始处偏移量为 4 的地方。(我们称表中的第一个元素为 arg(0)，当用户将其敲入命令行时，它指向程序的名字。)这就是为什么我们要将[EBX+4]位置处的内容压入堆栈的原因；我们已经将参数指针表的地址加载到了 EBX 中。位于[EBX+4]处的是指向 arg(1)——第一个命令行参数——的指针。如果仍然觉得有些糊涂的话，请参照图 12-4。

12.9.2 创建和打开文件

此时此刻，我们应该能够对于从汇编中调用 C 库的一般机制应付自如了。并且无论是否意识到这一点，我们已经能够相当轻松地应对一些文本文件的操纵机制了。我们已经通过标准输出的方式使用 printf()将格式化文本显示到显示器上。同样的机制也可用于将格式化文本写到基于磁盘的文本文件中——基本上将标准输出替换为真正的磁盘文件即可，所以理解文本文件 I/O 不应该算作一个概念上的飞跃。

但是与标准输出不同的是，它是为 C 库预定义的，而且始终可用，为了使用

基于磁盘的文本文件，我们必须创建或者打开它。fopen()函数就可以做这项工作。

有三种打开文件的一般方式：读取、写入和追加。当以读取的方式打开一个文件时，我们能够通过诸如 fgets()之类的函数从中读取文本，但是不能对该文件执行写入操作。当以写入的方式打开一个文件时，之前在文件中出现的任何内容都有可能被丢弃，新的材料从文件的开头开始写入。当以追加的方式打开一个文件时，可以对该文件执行写入操作，但是新的材料将被写入任何现有内容的后面，文件中的任何原来内容都会保留。

一般来说，当以写入的方式打开一个文件时，不能对其进行读取操作，但是有一些特定的模式，允许对一个文件既能执行读取操作，又能执行写入操作。对于文本文件，特别是(也是我们在这里正在谈论的那些)引入一些复杂性的文本文件，以及到目前为止的大多数文本文件而言，当打开时，要么用于读取，要么用于写入，但是不要在一次打开中同时允许两种操作。

在 Unix 文件系统中，如果以写入或者追加的方式打开一个尚未存在的文件，那么该文件将被创建。如果不知道一个文件是否已经存在，并且需要找到答案，尝试以读取的方式打开它，而不要以写入的方式，否则将肯定会得到这个文件，不管之前它是不是已经存在！

要想使用 fopen()，在调用之前，必须将以下参数压入堆栈：

(1) 首先压入堆栈的是一个用于指明应该采用哪种模式打开该文件的代码的地址。表 12-3 中已经列出了各种可用模式，对于文本文件而言，通常会使用的是“r”、“w”和“a”。它们应该被定义为由短字符构成的字符串，后面再跟一个空值：

```
WriteCode db 'w',0
OpenCode  db 'r',0
```

(2) 下一个进入堆栈的是包含即将打开的文件的名字的字符串的地址。

将这两项压入堆栈之后，就可以进行 fopen()函数调用了。如果文件被成功打开，fopen()函数将返回一个文件句柄到 EAX 中。文件句柄是在调用 fopen()函数时，Linux 分配给一个文件的 32 位数字。如果打开不成功，EAX 中的值将为 0。以下是以读取的方式打开一个文件的代码：

```
push opencode       ; 将“以读取的方式打开”的代码“r”的地址压入堆栈
push ebx            ; 帮助文件的名字的地址是存放在 EBX 中
call fopen          ; 尝试以读取的方式打开该文件
add esp,8           ; 堆栈清理：2 个参数*4 个字节 = 8
```

```
cmp eax,0          ; 如果打开操作失败，fopen()返回一个空值。
<根据需要进行跳转>
```

创建一个文件，然后对其进行写入操作的过程完全相同，除了一定要将“w”代码而不是“r”代码压入堆栈之外。

表 12-3　fopen()函数使用的文件访问代码

代码	描述
"r"	打开一个已经存在的文本文件，用于读取操作
"w"	创建一个新的文本文件或者打开和截断一个已经存在的文件
"a"	创建一个新的文本文件或者打开一个现有的文件，以便新的文本可被添加到文件的结尾
"r+"	打开一个现有的文本文件,要么用于写入操作，要么用于读取操作
"w+"	创建一个新的文本文件或者打开和截断一个现有的文件，同时具有读取和写入权限
"a+"	创建一个新的文本文件或者打开一个现有的文件，用于读取和写入，以便新的文本可以被添加到文件的结尾

12.9.3　使用 fgets()从文件中读取文本

当 fopen()函数成功地创建或者打开了一个文件时，它返回一个文件句柄到 EAX 中。注意，一定要将该文件的句柄存放到某个安全的地方——我建议要么将其复制到一个为此目的而分配的内存变量中，要么将它放到一个神圣的寄存器中。如果将其存储在 EAX、ECX 或者 EDX 中，然后对任一 C 库函数进行调用，那么位于这些寄存器中的文件句柄将被破坏，我们将会失去它。

一旦将一个文件以读取的方式打开，我们就能够使用 fgets()函数从中顺序读取文字行了。每次对一个打开的文本文件调用 fgets()，它都从该文件中读取一行文本，行被定义为一直到下一个 EOL(“换行”)字符(ASCII 码 10)处的所有字符，在 Unix 世界中，EOL 总是表示一个文本行的末尾。

现在，对于任意给定的文件而言，我们都没有办法知道到下一个换行符之前将有多少个字符，所以，如果放纵 fgets()去带回字符直到遇到一个换行符，将会很危险。如果试图打开一个错误类型的文件(二进制代码文件是一种可能性，再就是压缩数据文件)，在遇到一个文件系统认为是换行符字符的二进制值 10 之前，可能已经带进了成千上万个字节。任何分配用于保存传入文本的缓冲区都会

溢出，fgets()很可能会破坏相邻的数据，或者使程序崩溃。

基于这个原因，我们必须向 fgets()传递一个极限值。当开始阅读一行时，fgets()跟踪它已经从文件中带来了多少个字符，当到达比极限值少 1 时，停止读取字符。然后，增加一个 EOL 字符到缓冲区中，将其作为最后的字符并返回。

以这种方式建立对 fgets()的调用：

(1) 将从 fopen()函数返回的文件句柄压入堆栈。

(2) 将该字符计数极限值压入堆栈。这必须是一个实际的整数值，而不是一个指向该值的指针！

(3) 最后，将用于存放 fgets()从文件中读出的字符的字符缓冲区的地址压入堆栈。

所有工作完成后，调用 fgets()。如果 fgets()返回一个 0 到 EAX 中，那么，或者已经达到了该文件的结尾，或者可能在读取过程中发生了文件错误。无论哪种情况，都不会有更多的数据从文件中出来；如果 EAX 中没有返回一个 0 值，我们可以假设在传递给 fgets()的(存放在堆栈上的)地址处的缓冲区中已经出现了有效文本。

我曾经使用过 fgets()为 textfile.asm 创建一个非常简单的基于磁盘的帮助系统。当用户根本没有输入任何命令行参数时，textfile 程序从磁盘中读取一个简短的文本文件并将其显示到标准输出。如果这个基于磁盘的帮助文件无法打开，文本文件将显示一个大意是这种效果的短消息。这是一件命令行程序所需的、常见的、礼貌的事情，我建议为所有创建的日常实用工具都采用这种方式。

帮助系统的代码相对来说比较简单，并且同时演示了 fopen()函数和 fgets()两个函数的使用：

```
diskhelp:
    push OpenCode          ; 将指向“以读取方式打开”的代码“r”的指针压入堆栈
    push ebx               ; 指向帮助文件的名字的指针被传递到 ebx 中
    call fopen             ; 尝试打开文件，用于读取操作
    add esp,8              ; 清理堆栈
    cmp eax,0              ; 如果尝试打开操作失败，fopen 返回空值
    jne .disk              ; 从磁盘文件中读取帮助信息
    call memhelp           ; 此外，如果在磁盘上有帮助文件，从内存中读取
    ret
.disk: mov ebx,eax         ; 将打开文件的句柄保存到 ebx 中
.rdln: push ebx            ; 将文件的句柄压入堆栈
    push dword HELPLEN     ; 限制读取文本行的长度
    push HelpLine          ; 将帮助文本行缓冲区的地址压入堆栈
```

```
    call fgets              ; 从文件中读入一行文本
    add esp,12              ; 清理堆栈
    cmp eax,                ; 如果返回空值，表明发生了错误，或者到达了文件的结尾
    jle .done               ; 如果我们发现 eax 中的值为零，关闭并返回
    push HelpLine           ; 将帮助行的地址压入堆栈
    call printf             ; 调用 printf 显示帮助行
    add esp,4               ; 清理堆栈
    jmp .rdln
.done: push ebx             ; 把即将关闭的文件的句柄压入堆栈
    call fclose             ; 将句柄存放在堆栈上的那个文件关闭
    add esp,4               ; 清理堆栈
    ret                     ; 返回
```

当调用 diskhelp 过程时，主调程序将一个指向即将读入的帮助文件的名字的指针传递到 EBX 寄存器中。然后该代码尝试打开这个文件。如果尝试打开帮助文件失败，将显示一个来自存放在程序的.data 段的字符串的简短“安全故障”帮助信息。(这是对于 memhelp 的调用，它是 textfile.asm 程序中的另外一个较短的过程。)不要让用户盯着哑巴光标，猜测正在发生着什么事情！

一旦打开了帮助文件，代码将开始循环通过一个序列，该序列使用 fgets()从打开的文件中读取文本行，然后通过 printf()将这些行写入标准输出。即将被读取的行的最大长度由 HELPLEN 等值进行定义。将这个等值压入堆栈的操作与将立即数压入堆栈的操作没有任何不同，示例中的那条指令就是采用这种方法进行编码的。但是，不是在整个源代码中几个地方指定(也许有所不同)，帮助文件的行的最大长度只在一个地方定义，然后，只修改那个等值一次，就可以使程序中所有用到该等值的地方得到改变。等值非常好用。凡是能用它们的地方都要尽量使用它们。

每从文件中读取一行，该行的地址都被压入堆栈，并且通过 printf()进行显示。当即将被读取的帮助文件中没有更多的行时，fgets()返回一个零值到 EAX 中，程序跳转到关闭该文件的函数调用分支。

注意，fclose()函数的使用很简单：将一个打开文件的文件句柄压入堆栈，然后调用 fclose()。这就是关闭该文件所需的一切！

12.9.4 使用 fprintf()写文本到文件中

在本章的前面部分，我解释了如何使用 printf()函数通过标准输出的方式将格式化文本写入显示器。标准 C 库提供了一个函数，它能够将同样的格式化文本写

入任何打开的文本文件中。fprintf()函数能够做 printf()能做的所有事情，但是它需要一个放在堆栈上的额外参数：已打开的文本文件的文件句柄。printf()发送到标准输出的同样文本流将被 fprintf()发送到那个打开的文件。

至于如何使用格式化代码和基字符串来为 printf()格式化文本，这里就不再重复了。它们采用同样的方式，使用完全相同的代码。取而代之的是，我将会简单地总结一下如何设置对于 fprintf()的调用：

(1) 将任何值或者值的地址(必要时)压入堆栈。它的调用方式与 printf()的调用方式没有什么不同。

(2) 将包含格式代码的基字符串的地址压入堆栈。同样，它的完成方式与 printf()相同。

(3) 最后，这里是 fprintf()与 printf()的不同之处，将文本应该写入的目标文件的句柄压入堆栈。

然后调用 fprintf()。文字将被写入那个打开的文件中。请注意，要想使用 fprintf()，目标文件必须已被以写入的方式或者追加的方式打开。如果尝试对于一个以读取方式打开的文件使用 fprintf()函数，fprintf()将生成一个错误并且返回，根本不写入任何数据。

在那种情况下，将有一个错误编码返回到 EAX 中。但是，与到目前为止我们已经讨论过的其他函数不同，该错误代码是一个负数，而不是 0！所以，尽管应该将返回值与 0 相比较，实际上应该当一个值小于 0 时，而不是 0 本身时才跳转。通常情况下，要想在发生一个 fprintf()错误的情况下进行跳转，应该使用 JL(如果小于则跳转)，当遇到一个小于 0 的值时，它将跳转。

下面是来自 textfile.asm 程序的对 fprintf()的调用：

```
push esi                    ; ESI 是指向文本行的指针
push 1                      ; 将第一行的行号压入堆栈
push WriteBase              ; 将基字符串的地址压入堆栈
push ebx                    ; 将打开文件的文件句柄压入堆栈
writeline:
   cmp dword edi,0          ; 行计数器已经变为零了吗?
   je donewrite             ; 如果这样，往下走并清理堆栈
   call fprintf             ; 将文本行写入文件
   dec edi                  ; 递减即将被写入的行的计数值
   add dword [esp+8],1      ; 更新堆栈上的行号
   jmp writeline            ; 回去继续进行下一轮循环.
donewrite:
   add esp,16               ; 当完成 fprintf 调用后，清理堆栈
```

调用 fprintf()仅仅是相当小的一部分，但是这里还是有一些非常有趣的内容需要注意：调用了 fprintf()之后，代码并没有立即清理堆栈。在每一个对于 C 库函数的其他调用案例中，我们都调整了堆栈，使之在函数调用后立即删除该参数。这里有什么不同呢？

来自 textfile.asm 的这部分代码重复地将一个单独的文本行写到输出文件中，重复的次数由内存变量 LineCount 来指定。请记住，我们调用的函数并不从堆栈中删除参数，而且几乎从不修改它们。所以，不是浪费时间为每一次写入文件压入和移除参数，我一直在等待，直到对 fprintf()的所有调用都执行完毕，只有那时(在标签 donewrite 处)我才清理堆栈。

但是，这给我们留下了一个问题：在每次写入操作时都需要改变行号。在每一行文本被写入文件之前，textfile 程序都写入一个初始行号，而且那个行号对于每一个写入的行都发生变化。但是，不是为每一次 fprintf()调用都压入一个新的行号值到堆栈(这也将要求重新移除和重新压入其他的所有内容)，该代码在每次调用 fprintf()之后，直接进入堆栈，并更新已被压入堆栈的行号值：

```
add dword [esp+8],1          ; 更新堆栈上的行号
```

我对传递给 fprintf()的每个参数以字节为单位进行计数，并计算出压入的行号值位于堆栈中的什么位置。在本示例中(它可能会根据传递给 fprintf()的值的多少而有所不同)，这是一个比堆栈指针 ESP 高 8 个字节的位置。

修改已压入堆栈的参数已经不再是什么冒险，特别是如果它可以节省大量的压栈和弹出操作的话。我们只需确保精确地知道要想改变的内容在什么位置即可！不用说了，如果试图更新一个计数器的值，却取而代之为修改一个地址，将肯定导致程序快速崩溃。这是汇编语言，朋友。我们再也不能像在其他语言中一样轻松了。

12.9.5 关于收集过程到库中的注解

下面是一个关于如何收集过程到库中的概述：

- 不是必须得进行入口点定义或者寄存器保存。只要创建一个新的源代码文件，并将过程的源代码复制到该文件中，然后确保该文件必须以.asm为扩展名即可。
- 将在库中定义的所有过程的可调用入口点，以及任何可能被其他过程和

库使用的标识符定义为 global。

- 如果过程需要调用任何 C 库函数或者任何位于你拥有或创建的其他库中的过程，或者使用在库外定义的变量或其他标识符，将所有的这类外部标识符声明为 extern。
- 当从程序中调用库中的过程时，更新该程序的生成文件，以便最终的可执行程序拥有一个对于该库的依赖条件。

最后一点是唯一一个需要进一步讨论的内容。下面的生成文件将生成 textfile.asm 演示程序，该程序连接到一个名为 linlib.asm 的库。请注意，这里有一个全新的行用以指明目标文件 linlib.o 如何编译，并指明最后的二进制文件 textfile 既依赖于 textfile.o，也依赖于 linlib.o。

由于 textfile 的可执行文件依赖于 textfile.o 和 linlib.o，任何时候对 textfile.asm 或 linlib.asm 进行修改，生成实用工具都将通过 gcc 彻底重新连接可执行文件。但是，除非这两个.ASM 文件同时改变，否则只有那个被修改的.asm 文件被编译。生成的神奇之处就是它从来不作无用功。

```
textfile: textfile.o linlib.o
    gcc textfile.o linlib.o -o textfile
textfile.o: textfile.asm
    nasm -f elf -g -F stabs textfile.asm
linlib.o: linlib.asm
    nasm -f elf -g -F stabs linlib.asm
```

在本书的清单存档中有完整的 linlib.asm 文件。它所包含的过程在本章展示的其他程序已经体现，因此如果将它们都在这里重新打印，会显得重复。

最后，textfile.asm 程序的全部内容如清单 12-6 所示。请确保你可以阅读它的所有内容——这里没有本书尚未涵盖的内容。如果想要迎接挑战，这里是下一个项目：修改 textfile，使之读入一个文本文件，并将其再次写出，要求在每一行文本的前面都有一个行号前缀。同时允许用户在命令行中输入一个新的文件名用于保存修改后的文本。保留帮助系统，并为其编写一个新的帮助文本文件。

清单 12-6：textfile.asm

```
; 源文件名        : TEXTFILE.ASM
; 可执行程序名    : TEXTFILE
; 版本            : 2.0
; 创建日期        : 11/21/1999
; 最后更新日期    : 5/29/2009
```

```
; 作者            : Jeff Duntemann
; 描述            : 一个为 Linux 编写的文本文件输入/输出演示程序,使用 NASM 2.05
;
; 采用以下命令生成:
; nasm -f elf -g -F stabs textfile.asm
; nasm -f elf -g -F stabs linlib.asm
; gcc textfile.o linlib.o -o textfile
;
; 注意，这个程序需要一个名为 LINLIB.ASM 的外部程序中的几个过程。

[SECTION .data]         ; 包含已初始化数据的段

IntFormat dd '%d',0
WriteBase db 'Line #%d: %s',10,0
NewFilename db 'testeroo.txt',0
DiskHelpNm db 'helptextfile.txt',0
WriteCode db 'w',0
OpenCode db 'r',0
CharTbl db '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz-
@'
Err1 db 'ERROR: The first command line argument must be an
integer!',10,0
HelpMsg db 'TEXTTEST: Generates a test file. Arg(1) should be the
#  of ',10,0
HELPSIZE EQU $-HelpMsg
      db 'lines to write to the file. All other args are
         concatenated',10,0
      db 'into a single line and written to the file. If no text
         args',10,0
      db 'are entered, random text is written to the file. This
         msg ',10,0
      db 'appears only if the file HELPTEXTFILE.TXT cannot be
         opened. ',10,0
HelpEnd dd 0

[SECTION .bss]              ; 包含未初始化数据的段

LineCount  resd 1           ; 保留一个整数用于存放行号
HELPLEN    EQU 72           ; 定义帮助文本数据的一行的长度
HelpLine   resb HELPLEN     ; 为基于硬盘的帮助文本行保留空间
BUFSIZE    EQU 64           ; 定义文本行缓冲区 buff 的长度
```

```
Buff        resb BUFSIZE+5 ; 为一行文本保留空间

[SECTION .text]           ; 包含代码的段

;; 这些外部函数都来自 glibc 的标准 C 库：
extern fopen
extern fclose
extern fgets
extern fprintf
extern printf
extern sscanf
extern time

;; 这些外部函数均来自相关库 linlib.asm:
extern seedit            ; 为随机数产生器赋予一个种子
extern pull6             ; 产生一个 0 到 63 之间的 6 位随机数
extern newline           ; 输出指定数量的换行符字符

global main              ; 此代码是必须要有的，这样连接器才能找到入口点。

main:
    push ebp             ; 为调试器建立堆栈帧
    mov ebp,esp
    push ebx             ; 程序必须保存 EBP、 EBX、 ESI 和 EDI
    push esi
    push edi
;;; 在此之前的所有内容都是样板代码;请在所有的一般应用中使用它！
    call seedit          ;为随机数产生器赋予一个种子

;; 第一个测试是为了判断到底有没有命令行参数。
;; 如果一个也没有，我们将使用若干行显示帮助信息。
;; 不要忘记，第一个参数总是该程序的名字，所以总是至少有一个命令行参数！
    mov eax,[ebp+8]         ; 从堆栈中加载参数数量到 EAX 中
    cmp eax,1               ; 如果数量为 1,则没有参数
    ja chkarg2              ; 如果数量大于 1,继续

mov ebx,DiskHelpNm          ; 将帮助文件的名称的地址放入 ebx 中
    call diskhelp           ; 如果只有一个参数,则显示帮助信息
    jmp gohome              ; 并且退出该程序

;; 下一步，我们检查数字型命令行参数 1:
```

```
chkarg2:
    mov ebx,[ebp+12]          ; 将指向参数表的指针放入 ebx 中
    push LineCount            ; 为了调用 sscanf，将表示行数的整数的地址压入堆栈
    push IntFormat            ; 将整数格式化代码的地址压入堆栈
    push dword [ebx+4]        ; 将指向参数(1)的指针压入堆栈
    call sscanf               ; 调用 sscanf 将参数(1)转换成一个整数
    add esp,12                ; 清理堆栈
    cmp eax,1                 ; 返回值为 1，说明我们得到了一个数字
    je chkdata                ; 如果我们得到了一个数字，继续；否则异常中断
    mov eax,Err1              ; 将错误信息#1 的地址加载到 eax 中
    call showerr              ; 显示出错信息
    jmp gohome                ; 退出程序

;; 这里我们正在查看是否还有更多的参数。如果有，我们将它们连接到一个
;; 大小不超过 BUFSIZE 个字符的单个字符串中。(是的，我知道这些代码是在做 strncat
;; 所做的事情)
    chkdata:
    cmp dword [ebp+8],3       ; 还有第二个参数吗?
    jae getlns                ; 如果这样，说明我们有了一个即将填充到一个文件中的文本
    call randline             ; 如果没有，产生一行随机文本行
                              ; 注意，randline 返回指向行的指针到 esi 中
    jmp genfile               ; 继续创建该文件

;; 这里，我们将所有的命令行文本，最大到 BUFSIZE 个字符，复制到行缓冲区 buff 中。
;; 我们跳过(此时我们知道已经存在的)前两个参数，但是我们知道至少有一个
;; 文本参数在 arg(2)中。进入本段内部，我们知道 ebx 包含着指向参数表的指针。
;; 所有其他寄存器的值都不可预料。
getlns: mov edx,2             ; 我们知道至少有 arg(2)，从那里开始
    mov edi,Buff              ; 目标指针是字符缓冲区的起点
    xor eax,eax               ; 为字符计数器将 eax 清零，
    cld                       ; 为以“上内存”的方式执行 movsb 而清零方向标志

grab: mov esi,[ebx+edx*4]       ; 将指向下一个参数的指针复制到 esi 中
.copy: cmp byte [esi],0         ; 我们已经找到了该参数的结尾了吗?
       je .next                 ; 如果这样的话，接着处理下一个参数
       movsb                    ; 从[esi]复制字符到[edi]；递增 edi 和 esi
       inc eax                  ; 将字符总数计数值递增
       cmp eax,BUFSIZE          ; 判断一下是否缓冲区已经填充了最大数量的字符
       je addnul                ; 如果是这样的话，添加一个空值到 buff，
                                ; 并且我们的任务已经完成

       jmp .copy
```

```
.next: mov byte [edi],' '       ; 复制空格到 buff 中以便将参数分开
       inc edi                  ; 为空格递增目标指针
       inc eax                  ; 同时将字符计数器加 1
       cmp eax,BUFSIZE          ; 判断我们是否已经填满了 buff
       je addnul                ; 如果这样，继续往下进行，添加一个空值，
                                ; 并且我们的任务已经完成
       inc edx                  ; 否则，递增参数计数器
       cmp edx, dword [ebp+8]   ; 与参数个数相比较
       jae addnul               ; 如果 edx = 参数个数 ，我们完成了任务
       jmp grab                 ; 然后回去，复制它

addnul: mov byte [edi],0        ; 在 buff 的结尾附加一个空值
       mov esi,Buff             ; 文件写代码期望指向文本的指针放入 esi 中

;; 现在，我们创建一个文件来存放拥有的文本：
genfile:
       push WriteCode           ; 将指向文件写入/创建代码('w')的指针压入堆栈
       push NewFilename         ; 将指向新文件名的指针压入堆栈
       call fopen               ; 创建/打开文件
       add esp,8                ; 清空堆栈
       mov ebx,eax              ; 将 eax 中包含的文件句柄保存在到 ebx 中

;;文件是打开的。现在让我们向文件中填充文本：
       mov edi,[LineCount]      ; 即将被填充文本的行数存放在 edi 中
       push esi                 ; ESI 是指向文本行的指针
       push 1                   ; 第一个文本行
       push WriteBase           ; 将基字符串的地址压入堆栈
       push ebx                 ; 将那个打开文件的句柄压入堆栈

writeline:
       cmp dword edi,0          ; 文件计数器已经到达 0 了吗？
       je donewrite             ; 如果是这样的话，往下走并清理堆栈
       call fprintf             ; 将文本行写入到文件中
       dec edi                  ; 递减即将被写入的行的计数器
       add dword [esp+8],1      ; 更新堆栈上的行数
       jmp writeline            ; 回去继续循环，再次重复做这件事
donewrite:
       add esp,16               ; 在进行 fprintf 调用之后，清理堆栈
;; 我们已经完成了文本写入任务；现在让我们关闭那个文件：
closeit:
```

```
        push ebx                ; 将即将被关闭的文件的句柄压入堆栈
        call fclose             ; 关闭句柄存放在堆栈上的那个文件
        add esp,4
;;; 至此以后的所有内容都是样板代码;请在所有的一般应用程序中使用它!
        gohome: pop edi             ; 恢复保存的寄存器
        pop esi
        pop ebx
        mov esp,ebp                 ; 在返回之前，销毁堆栈帧
        pop ebp
        ret                         ; 将控制返回给 C 关闭代码

;;; 子函数==================================================
;--------------------------------------------------------------

; 基于磁盘的小型帮助子程序——最后修改日期: 12/5/1999
; 这个例程从一个文本文件中读入文本，文本文件的名字通过一个指向该名字字符串的、
; 存放在 ebx 中的指针来进行传递。该例程打开那个文本文件，从中读入文本，然后将这
; 些文本显示到标准输出中. 如果文件不能被打开，取而代之的是显示一个非常简短的基
; 于内存的消息
;--------------------------------------------------------------

diskhelp:
        push OpenCode       ; 将指向“打开用于读取”的代码“r”的指针压入堆栈
        push ebx            ; 指向帮助文件名的指针被传递到 ebx 中
        call fopen          ; 尝试以读取的方式打开文件
        add esp,8           ; 清理堆栈
        cmp eax,0           ; 如果尝试打开操作失败，fopen 返回空值，
        jne .disk           ; 从磁盘中读入帮助信息，否则从内存中读入信息
        call memhelp
        ret
.disk:  mov ebx,eax         ; 将打开的文件的句柄保存到 ebx 中
.rdln:  push ebx            ; 将文件句柄压入堆栈
        push dword HELPLEN ; 限制读入文本行的长度
        push HelpLine       ; 将帮助文本行缓冲区的地址压入堆栈
        call fgets          ; 从文件中读入一行文本
        add esp,12          ; 清理堆栈
        cmp eax,0           ; 如果返回一个空值，则表明错误或者到达了文件末
        jle .done           ; 如果 eax 中的值为零，停止并返回
        push HelpLine       ; 将帮助行的地址压入堆栈
        call printf         ; 调用 printf 来显示帮助行
        add esp,4           ; 清理堆栈
```

```
        jmp .rdln

.done: push ebx             ; 将即将被关闭的文件的句柄压入堆栈
        call fclose         ; 将句柄存放在堆栈上的那个文件关闭
        add esp,4           ; 清理堆栈
        ret                 ; 返回

memhelp:
        mov eax,1
        call newline
        mov ebx,HelpMsg          ; 将帮助文本的地址加载到 eax 中

.chkln: cmp dword [ebx],0    ; 帮助消息指针指向一个空值了吗?
        jne .show                ; 如果没有,显示帮助行
        mov eax,1                ; 将即将输出的换行符的数量加载到 eax 中
        call newline             ; 输出换行符
        ret                      ; 如果执行完毕，则返回

.show: push ebx                  ; 将帮助行的地址压入堆栈
        call printf              ; 显示该行
        add esp,4                ; 清理堆栈
        add ebx,HELPSIZE         ; 将地址递增帮助行的长度个字节
        jmp .chkln               ; 回去继续循环，检查一下我们是否已经完成任务

showerr:
        push eax                 ; 在入口处， eax 包含错误信息的地址
        call printf              ; 显示错误信息
        add esp,4                ; 清理堆栈
        ret                      ; 返回; 没有返回值

randline:
        mov ebx,BUFSIZE          ; BUFSIZE 告诉我们需要拉入多少个字符
        mov byte [Buff+BUFSIZE+1],0   ; 首先在缓冲区的结尾放入一个空值

.loop: dec ebx                   ; BUFSIZE 是基于 1 的，所以递减
        call pull6               ; 获取一个 0 到 63 之间的随机数
        mov cl,[CharTbl+eax]     ; 将 eax 中的随机数# 用作表中的偏移量

;并将字符从表中拷贝到 cl 中
        mov [Buff+ebx],cl        ; 将字符从 cl 中复制到字符缓冲区中
        cmp ebx,0                ; 我们已经兴致勃勃地完成了任务吗?
```

```
        jne .loop              ; 如果没有，回去并进行下一轮循环
        mov esi,Buff           ; 将缓冲区的地址复制到 esi 中
        ret                    ; 并且返回
```

成功完成所有内容之后，可以圆满地谢幕了，因为你已经成为一个真资格的汇编语言程序员！

结论：不是结束，而是刚刚开始

你从来没有真正学会汇编语言。

随着时间的推移，你可以通过阅读一些关于这一主题的好书、阅读一些别人已经编写的不错的代码，而且，最重要的是，通过自己编写很多很多代码，来不断提高自己的技能。但是，你绝对不能对外宣称，“我学会了汇编语言。”

不要为此感到难过。事实上，我偶尔听说迈克尔•亚伯拉什(《汇编语言禅宗》、《代码优化禅宗》以及巨著《迈克尔•亚伯拉什的图形编程黑皮书》的作者)已经学到了一些关于汇编语言的新东西，这个消息使我倍受鼓舞。迈克尔已经编写了将近 30 年的高性能汇编代码，并且已经成为整个西半球中两三个顶尖汇编语言程序员之一。

如果迈克尔仍然在学习，那么我们其他人会有希望学会汇编语言吗？

错误的问题。愚蠢的问题。如果迈克尔仍然在学习，这意味着我们全部都是学生，而且将来也永远是学生。这就是说，旅程才是目标，只要我们继续探索、开辟、研究和尝试从来没有尝试过的事情，随着时间的推移，我们会不断提高技术水平，我们编写的程序也将越来越让那些行业前辈们刮目相看。

问题不在于征服，而是与它一起成长，不断发展这方面的相关知识。由于旅程是目标，所以在本书中，我尽量努力帮助那些一直盯着汇编语言的复杂性，不知敲门砖在何处，一想到开始学习之旅程就惶惶然的人们。

诀窍很简单，只要相信自己就可以了。

你能行的！我们面临的挑战不仅限于汇编语言。考虑一下我的个人经历：我于经济衰退的 1974 年离开学校，并获得了英语方向的文学学士学位，成绩优异，但是除了开出租车之外，我很少有机会拥有可靠的前景。我竭尽全力获取了一份为施乐公司修理复印机的工作。书是有趣的，但是文书工作能够赚钱——所

以，在调整工作路线，转行做计算机程序员之前，我拿起工具袋，度过了几年的愉快时光。

但是，我永远不会忘记当我从一名技术人员肩头向后望去的第一个可怕时刻，我看到一台已被拿下操作面板的 model 660 复印机，它看起来就像是一个由小齿轮，凸轮、磁鼓以及那些用于不断转动、翻转和前后敲打开关驱动器的扣链齿轮组成的无底洞。我被复杂性迷惑了，忘记了注意一张纸已被放入机器，变成了原来文件的副本。我因为从来不知道那些令人害怕的小凸轮是做什么的而错过了令人倍感欣慰的大局上的简单性——是复印机在复印。

从总体上看，这是一个正方形的家伙。先暂时忽略凸轮和齿轮。你一定能做到的。在开始观察那些凸轮和齿轮前，先找出整体中的重要部分(如果不是很明显的话，可以请教别人)并且研究它。探明发生的过程。然后再将大图划分为小图。看看情况是如何发展的。只有那时，你才应该注意一些淹没在混乱中的小东西，比如单独的凸轮或开关。

这就是你征服复杂性的利器，也是我已在《Linux 与汇编语言基础》这本书中所呈现的办法。有人可能会说，我已经对指令集偷工减料了，但是，全面介绍指令集从来都不是本书的真正目标。真正的目标是通过一些隐喻和大量的图片以及轻松的心情(这一点真的很重要！)，帮助你克服对这一主题的复杂性的恐惧。

这能否奏效呢？还是让你来给出答案吧！我真的很想知道。

进一步学习

人们有时会生我的气，因为这本书(自 1989 年以已经差不多有四个版本)讲解得不够深入。但是我的立场非常坚定：这本书是针对初学者的，我不会为了在末尾加上更多的材料而减少开头的内容。(书只能这么长！)为了进行更深一步的研究，你需要将这本书放到一边，靠自己的努力去继续修行了。

一般方法应该是这样的：

- 学习 Linux。
- 学习汇编语言。
- 编写代码。
- 编写更多的代码。

在 Linux 方面从来没有缺少过好书。下面是我推荐的一些书籍。

Ubuntu 8.10 Linux Bible，作者：William von Hagen(Wiley , 2009)

如果你还没有充分了解 Linux，这是一本值得拥有的好书。虽然根本不涉及编程，但是它涵盖了几乎所有与 Linux 使用有关的内容。即使你已经在 Linux 方面学的相当好，当意识到有某一特定主题，但就是想不起来具体内容时，手头有一本索引良好的参考书总是很方便的。

Beginning Linux Programming，第四版，作者：Neil Matthew 和 Richard Stones (Wrox Press, 2008)

如果你之前没有任何 Linux 编程基础，这本书提供了一个很好的概述。它没有讨论汇编语言，而是聚焦于 shell 编程、C/C++和 SQL。不过，那些与文件编程、终端，以及 Linux 环境有关的内容适用于任何类型的编程，而且比我在这本书中讲得还要深入。

关于汇编语言方面的书籍历来不是很多，好书更是凤毛麟角。许多关于这一主题的书籍已经很老了，并且现在大多数已被废弃了，比如 Tom • Swan 的*掌握 Turbo 汇编*(Sams，1995 年)，主要包括 DOS，同时也包含少部分与 Windows 相关的内容。

The Art of Assembly Language，作者：Randall Hyde (No Starch Press, 2003)

纯粹就质量而论，这可能是写得最好的汇编语言书籍。但是它有点奇怪，因为它描述了一种“高级”汇编编译器(HLA)，这种汇编编译器是由 Randy 自己编写的，它能理解一种包含某些从诸如 C 之类的高级语言中借过来的元素的语法。在汇编语言中引进高级语言元素使得在最底层发生的事情变得模糊，而这些又恰恰是学习汇编语言的目标所在，所以我犹豫不定。那就是说，HLA 应该像 C 语言那样：一种低级语言，提供对于单个指令的访问，同时允许使大型程序成为可能的高级数据抽象。如果你是一名熟练的 C 程序员，那么，HLA 提供了一条学习汇编语言的理想途径；但是如果你刚刚开始学习汇编语言，请暂时编写纯粹的汇编程序，直到你肯定自己已经理解了它，一直到超过最后一批海龟。

Professional Assembly Language，作者：Richard Blum(Wrox Press, 2005)

这是一本不错的中级书，同时也是你进入汇编语言的下一里程碑的不错选择，单是关于数字编程方面的 3 章就非常值得买一本了。(在本书所有内容中，我一直没有足够的篇幅来涵盖浮点运算。)这本书的一个较大的缺点是，作者主要采用 gas 汇编编译器和 AT&T 助记符，尽管 gas 有些乖僻，但是如果想理解大部分关于汇编方面的书籍和网页，仍需了解它和它的语法。之后的其他选择将使你收获甚微。在汇编方面还有一些其他书籍，我敢肯定，大部分我都看过，但是如果我不在这里提及它们，意味着出于某种原因我不推荐它们。不过，如果你发

现一本你喜欢且觉得有用的好书，请给我留言，让我知道。我很关注这个领域，但是我将会成为最后一个坚持从不遗漏任何事情的人。请通过我的汇编语言网页与我联系：

www.duntemann.com/assembly.htm

从头起跑

好了，手里拿着一摞新书，再美美地睡上一觉之后，你要开始独自奋斗了。为自己设定一个目标，并努力实现它：有些东西很棘手，比如：编写一个能够通过一个给定的含混不清的文件名在特定的目录树中定位所有文件的汇编语言实用工具。这对一个汇编新手而言，有些费劲，并且需要一些研究和探讨以及(也许)一些挫折。但你可以做到这一点，一旦做到了，你就可以成为真正的汇编语言程序员“熟练工”了。

专家也是需要工作和时间才能炼成的。书籍只能将你带到这么远。最终你必须成为自己的老师，主导自己的学习过程。目前，掌握汇编语言意味着理解操作系统内核和它的底层机制，比如设备驱动程序等。为了做到这一点，需要学好C，这没什么可说的。掌握汇编语言也越来越涉及编写那些运行在高性能图形协处理器(比如 Nvidia 系列的图形协处理器)上的代码。如今，游戏业驱动着性能计算，虽然编写高性能图形软件是一项非常困难的挑战，但是其结果也相当激动人心。

无论采取何种路线，都要坚持编程。不论编写什么样的代码，在编写那些导致新的挑战的代码期间，你都会学到一些东西。而学到一些新的东西，总会让你到认识到，还有很多的东西需要学习。学习是必要的，但是如果没有不断进取和持之以恒的实践，学习也不会给你带来任何帮助，静态的知识如果没有实际经验的巩固，很容易变得陈腐。

有时它会变得可怕。计算的复杂性似乎每两年就增加一倍。不过不要怕，一定要坚持告诉自己：我能做到的！相信此话的真理，便是离开起点的本质——而且其余的路，像所有的路一样，要脚踏实地地、一步一个脚印地走下去。

附录 A 部分 x86 指令集

关于指令集参考资料的说明

指令操作数

当一条指令使用两个操作数时，左侧为目标操作数，右侧为源操作数。一般来说，当指令生产一个结果时，这个结果将替换目标操作数。例如，在指令

```
ADD BX,SI
```

寄存器 BX 中的值将与寄存器 SI 中的值相加，然后将结果存入寄存器 BX 中，覆盖掉相加之前的 BX 中的任何值。

标志结果

每条指令的讲解中都包含一个如下所示的标志概要，(在不同的指令中星号的出现情况会有所不同)：

```
O  D  I  T  S  Z  A  P  C    OF: 溢出标志   TF:陷阱标志    AF: 辅助进位标志
F  F  F  F  F  F  F  F  F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*     ?  ?  *  *  *  *  *    IF: 中断标志   ZF: 零标志     CF: 进位标志
```

这里展示了 9 个最重要的标志。当标志的下方出现星号时，表明该页所描述的指令影响了这个标志。当标志的下方为空白时，表明该指令不会以任何方式影响这个标志。如果一个标志按照定义好的方式受到了影响，那么它遵从的是如下规则：

OF 如果结果太大，以至于在目标操作数中存放不下时，OF 被设置。

IF 被 STI 指令设置；被 CLI 清零。

TF　该标志是为调试器准备的；不用于一般的编程中，可以忽略。

SF　当计算结果强制目标操作数变为负数时，SF 被设置为 1。

ZF　当操作数的结果为零时被设置；如果结果不为零，则被清零。

AF　辅助进位标志，用于 4 位 BCD 码数学运算。当某个操作导致一个 4 位 BCD 量产生进位或借位时，AF 被设置。

PF　当结果的最低字节中值为 1 的位的个数是偶数时被设置；为奇数时被清零。用于数据通信，在其他领域用得很少。

CF　如果一个加法或移位操作的结果从目标操作数借位或进位，CF 被设置，否则被清零。如果 CF 在某一行动开始之前必须处于可知状态，可以使用 STC 来手动将 CF 设置为 1，使用 CLC 来手动将 CF 清零。

有些指令强制某些标志变为不确定状态。这种情况通过在对应的标志标头下面用一个问号来进行标识。“不确定”的意思是并不希望它位于某一特定状态。在知道它已经被另一条指令的行为从某一状态变为另一种定状态之前，不要测试或者以任何其他方式使用此类标志的状态。

AAA：BCD 码加法后调整 AL

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
?         ? ? * ? * IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
AAA
```

举例

```
AAA
```

说明

当两个 BCD 码而非普通的二进制值相加时，AAA 指令对出现在 AL 中的和进行调整。一定要注意，AAA 本身并不执行这个算术运算，它只是前面的 ADD 或者 ADC 指令的一个后期制作。AL 寄存器是一个隐式操作数，并且不可以被显式指明。所以，一定要确保 ADD 或者 ADC 指令将结果存放到 AL 中！

一个 BCD 码值是一个高四位被设置为零且低四位包含一个 0～9 之间的数的

字节。如果前面的 ADD 或者 ADC 指令处理一个或者两个大于 9 的操作数时，AAA 将产生垃圾结果。

在两个合法的 BCD 码相加之后，AAA 将一个非 BCD 结果(也就是说，一个 AL 中的比 9 大的结果) 调整为一个 0～9 之间的值。由此产生的进位叫作十进制进位，因为它是一个 BCD 码进位而不是简单的二进制码进位。

例如，如果 ADD 指令将 08 和 04 (都是合法的 BCD 码) 相加，将在 AL 中产生 0C，AAA 会取出 0C 并将其调整为 02。而十进制进位将存入 AH 中而不是 AL 的高 4 位中，AL 的高 4 位总是被 AAA 清零。

如果前面的 ADD 或者 ADC 指令导致了一个十进制进位(像上例中所说的那样)，CF 和 AF 都被设置为 1，并且 AH 中的值 被递增 1。否则，AH 中的值不递增，并且 CF 和 AF 都被清零。

这条指令很微妙。请到正文中去查阅详细的讲解。

```
r8  = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr  = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8  = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据               i8  = 8 位立即数
i16 = 16 位立即数                 i32= 32 位立即数
d8  = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

ADC：带进位的算术加法指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*       * * * * *  IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
ADC  r8,r8
ADC  m8,r8
ADC  r8,m8
ADC  r16,r16
ADC  m16,r16
ADC  r16,m16
ADC  r32,r32       386+
ADC  m32,r32       386+
```

```
ADC  r32,m32      386+
ADC  r8,i8
ADC  m8,i8
ADC  r16,i16
ADC  m16,i16
ADC  r32,i32      386+
ADC  m32,i32      386+
ADC  r16,i8
ADC  m16,i8
ADC  r32,i8       386+
ADC  m32,i8       386+
ADC  AL,i8
ADC  AX,i16
ADC  EAX,i32      386+
```

举例

```
ADC  BX,DI
ADC  EAX,5
ADC  AX,0FFFFH          ;使用单字节操作码
ADC  AL,42H             ;使用单字节操作码
ADC  BP,17H
ADC  WORD  [BX+SI+Inset],5
ADC  WORD  ES:[BX],0B800H
```

说明

ADC 将源操作数和进位标志与目标操作数相加，并且在操作之后，用结果替换目标操作数。该加法操作是算术加法，并且进位允许我们跨几个寄存器或多个内存位置进行多精度加法。(要想进行不考虑进位标志的加法，请使用 ADD 指令。) 所有受影响的标志都根据该操作的结果进行相应的设置。最重要的是，如果结果太大，以至于在目标操作数中存放不下时，进位标志将被设置为 1。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8  = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据             i8  = 8 位立即数
i16 = 16 位立即数               i32= 32 位立即数
d8  = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

ADD：算术加法指令

影响的标志

O	D	I	T	S	Z	A	P	C
F	F	F	F	F	F	F	F	F
*				*	*	*	*	*

OF：溢出标志　TF：陷阱标志　AF：辅助进位标志
DF：方向标志　SF：符号标志　PF：奇偶标志
IF：中断标志　ZF：零标志　CF：进位标志

合法形式

```
ADD r8,r8
ADD m8,r8
ADD r8,m8
ADD r16,r16
ADD m16,r16
ADD r16,m16
ADD r32,r32        386+
ADD m32,r32        386+
ADD r32,m32        386+
ADD r8,i8
ADD m8,i8
ADD r16,i16
ADD m16,i16
ADD r32,i32        386+
ADD m32,i32        386+
ADD r16,i8
ADD m16,i8
ADD r32,i8         386+
ADD m32,i8         386+
ADD AL,i8
ADD AX,i16
ADD EAX,i32        386+
```

举例

```
ADD BX,DI
ADD AX,0FFFFH        ;使用单字节操作码
ADD AL,42H           ;使用单字节操作码
ADD EAX,5
ADD BP,17H
ADD DWORD [EDI],EAX
ADD WORD [BX+SI+Inset],5
ADD WORD ES:[BX],0B800H
```

说明

ADD 将源操作数与目标操作数相加，并且在操作完成之后，用结果替换目标操作数。该加法操作是一个不考虑进位标志的算术加法。(要想使用进位标志，请使用 ADC——带进位相加指令。)所有受影响的标志都根据该操作的结果被设置。最重要的是，如果结果太大，以至于在目标操作数中存放不下时，进位标志将被设置为 1。

```
r8 = AL AH BL BH CL CH DL DH r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS       r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32= 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

AND：逻辑与指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志  TF: 陷阱标志  AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志  SF: 符号标志  PF: 奇偶标志
*       * * ? * *   IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式

```
AND r8,r8
AND m8,r8
AND r8,m8
AND r16,r16
AND m16,r16
AND r16,m16
AND r32,r32        386+
AND m32,r32        386+
AND r32,m32        386+
AND r8,i8
AND m8,i8
AND r16,i16
AND m16,i16
AND r32,i32        386+
AND m32,i32        386+
```

```
AND AL,i8
AND AX,i16
AND EAX,i32        386+
```

举例

```
AND BX,DI
AND EAX,5
AND AX,0FFFFH                 ;使用单字节操作码
AND AL,42H                    ;使用单字节操作码
AND DWORD [EDI],EAX
AND WORD ES:[BX],0B800H
AND WORD [BP+SI],DX
```

说明

AND 指令对它的两个操作数执行逻辑与操作。一旦操作完成，结果将替换目标操作数。与操作是按位执行的，源操作数的第 0 位与目标操作数的第 0 位相与，源操作数的第 1 位与目标操作数的第 1 位相与，以此类推。如果两个操作数同时为 1，与操作产生一个 1；仅当任一操作数为 0 时，与操作产生零。请注意，该操作使得辅助进标志变为不确定状态。CF 和 OF 被清零，而其他受影响的标志则根据操作的结果进行设置。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32= 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

BT：位测试指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
                *   IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
BT r16,r16         386+
```

```
BT m16,r16          386+
BT r32,r32          386+
BT m32,r32          386+
BT r16,i8           386+
BT m16,i8           386+
BT r32,i8           386+
BT m32,i8           386+
```

举例

```
BT AX,CX
BT [BX+DI],DX
BT AX,64
BT EAX,EDX
BT ECX,17
```

说明

BT 指令从左边的操作数中复制某一指定的位到进位标志中，在那里，我们可以通过使用一个移位/循环指令来测试它，或者将其反馈到一个量中。具体复制哪个位由右边的操作数指定。两个操作数都不会被 BT 改变。

当右边的操作数是一个 8 位的立即数时，该值指定了即将被复制的位的编号。在“BT AX,5”指令中，AX 的第 5 位被复制到 CF 中。当立即数超过了左边操作数的位数，该值被表示为对左边操作数位数取模。也就是说，因为在 EAX 中没有 66 个位，所以“BT EAX, 66”从该立即数中拿出 32 的最大倍数后，余下的即为待复制位的编号。(在这里，为 2。)当右操作数不是一个立即数时，该右操作数不仅指定了即将被测试的位，而且指定了相对于左侧操作数的内存引用的偏移地址。

这有点复杂，而且在本书中没有彻底涵盖。请到一个完整的汇编语言参考资料中查阅详细的解释。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8 = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

CALL：调用过程调用指令

影响的标志

```
O  D  I  T  S  Z  A  P  C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F  F  F  F  F  F  F  F  F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <无>                  IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
CALL <near label>
CALL <far label>
CALL r16
CALL m16
CALL r32   386+
CALL m32   386+
```

举例

```
CALL InsideMySegment      ;InsideMySegment 是一个近标号。
CALL OutsideMySegment     ;OutsideMySegment 是一个远标号。
CALL BX
CALL EDX
CALL WORD [BX+DI+17]      ;调用位于[BX+DI+17]处的近地址
CALL DWORD [BX+DI+17]     ;调用位于[BX+DI+17]处的完整的 32 位地址
```

说明

CALL 指令将 CPU 控制权移交到一个过程的地址处。在移交控制之前，CALL 将紧位于它本身后面的那条指令的地址压入堆栈。这允许一个 RET 指令将返回地址或者弹出到 CS:IP 中，或者只弹出到 IP 中(取决于它是一个近调用，还是远调用)，从而在 CALL 指令执行完毕后，能够将控制权立即返回给该指令。

除了能对一个已定义的标号进行显式调用之外，CALL 指令还能够将控制移交到一个位于某一 16 位通用寄存器中的近地址处，也可以移交给一个位于内存中的地址。这在合法形式列中分别用 m16 和 m32 表示。m32 就是一个存储在可以通过任意合法的 x86 内存寻址模式来寻址的内存中的某个位置处的完整的 32 位地址。指令 CALL m16 和指令 CALL m32 在创建“过程地址跳转表”时很有用。

还有很多 CALL 指令的变体，它们均工作在操作系统的保护机制之下。这些

在这里都没有涵盖；要想获取更多的信息，你应该阅读一个更加深入的文本或者一个完整的汇编语言参考资料。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

CLC：清除进位标志(CF)指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
                *    IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
CLC
```

举例

```
CLC
```

说明

CLC 只是将进位标志(CF)设置为清零(0)状态。如果在工作开始之前，要求进位标志必须处于已知的清零状态，请使用 CLC。例如：当你使用循环指令 RCL 和 RCR 来将一系列字或字节进行循环移位时。它也可被用于从一个程序返回之前将 CF 置于某一已知状态，以根据需要表明该程序执行成功还是失败。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

CLD：清除标志位(DF)指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志  TF: 陷阱标志  AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志  SF: 符号标志  PF: 奇偶标志
  *                  IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式

```
CLD
```

举例

```
CLD
```

说明

CLD 只是将方向标志(DF)设置为清零(0)状态。这将会影响由重复字符串指令，如 STOS、SCAS 和 MOVS，所执行的调整。通常情况下，当 DF= 0 时，目标指针增加，当 DF= 1 时,目标指针减小。 DF 可以通过 STD 指令设置为 1。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

CMP：算术比较指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志  TF: 陷阱标志  AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志  SF: 符号标志  PF: 奇偶标志
*       * * * * *    IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式

```
CMP r8,r8
CMP m8,r8
```

```
CMP r8,m8
CMP r16,r16
CMP m16,r16
CMP r16,m16
CMP r32,r32        386+
CMP m32,r32        386+
CMP r32,m32        386+
CMP r8,i8
CMP m8,i8
CMP r16,i16
CMP m16,i16
CMP r32,i32        386+
CMP m32,i32        386+
CMP r16,i8
CMP m16,i8
CMP r32,i8         386+
CMP m32,i8         386+
CMP AL,i8
CMP AX,i16
CMP EAX,i32        386+
```

举例

```
CMP BX,DI
CMP EAX,5
CMP AX,0FFFFH         ;使用单字节操作码
CMP AL,42H            ;使用单字节操作码
CMP BP,17H
CMP WORD [BX+SI+Inset],5
CMP WORD ES:[BX],0B800H
```

说明

CMP 比较它的两个操作数，并设置相关标志位以表明比较的结果。目标操作数不受影响。操作本身等同于从源操作数中减去目标操作数，不考虑借位(SUB)，只是结果不会替换目标操作数。通常情况下，CMP 后面跟有一条条件跳转指令，也就是说，JE 是为了实现如果操作数相等，则跳转，JNE 是为了实现如果它们不等则跳转，等等。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                 m16 = 16 位内存数据
```

```
m32 = 32 位内存数据          i8 = 8 位立即数
i16 = 16 位立即数            i32 = 32 位立即数
d8 = 8 位有符号位移          d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

DEC：递减操作

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*       * * * *     IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
DEC m8
DEC m16
DEC m32
DEC r8
DEC r16
DEC r32
```

举例

```
DEC AL
DEC CX
DEC EBX
DEC BYTE [BP]           ;递减位于[BP]处的字节
DEC WORD [BX]           ;递减位于[BX]处的字
DEC DWORD [EDX]         ;递减位于[EDX]处的双字
```

说明

请记住，DEC 指令不能递减段寄存器。所有的“半寄存器”操作码的长度为 2 个字节，但是所有的 16 位寄存器的操作码的长度为 1 个字节。如果你能够将只有低半部分包含数据的整个 16 位寄存器递减 1，那么请使用 16 位寄存器，这样，操作码可以节省一个字节。DEC 并不影响进位标志。(这是一个经常容易犯的错误。)

像所有能够操作内存的其他指令一样，内存数据形式必须与大小标识符一同使用，这些标识符有 BYTE、WORD 和 DWORD。NASM 并不假设使用任何一种大小标识符！

```
r8 = AL AH BL BH CL CH DL DH  r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS        r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

DIV：无符号整数除法指令

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	OF：溢出标志	TF：陷阱标志	AF：辅助进位标志
?				?	?	?	?	?	DF：方向标志	SF：符号标志	PF：奇偶标志
									IF：中断标志	ZF：零标志	CF：进位标志

合法形式

```
DIV r8
DIV m8
DIV r16
DIV m16
DIV r32     386+
DIV m32     386+
```

举例

```
DIV CH              ; AX / CH -->商数放在 AL 中; 余数放在 AH 中
DIV BX              ; DX:AX / BX -->商数放在 AX 中; 余数放在 DX 中
DIV ECX             ; EDX:EAX / ECX -->商数放在 EAX 中; 余数放在 EDX 中
DIV WORD [BX+DI]    ; DX:AX / [BX+DI] --> 商数放在 AX 中; 余数放在 DX 中
```

说明

DIV 将隐式被除数与在它的操作数中显式指定的除数相除。对于 8 位除数而言，被除数假定存在于 AX 中。对于 16 位和 32 位除数而言，被除数假定存在于两个寄存器中，从而允许执行更大范围内的计算。被除数的最低有效部分被放到“A ”系列寄存器(AX 的或的 EAX)中，最高有效部分被放到“D”系列寄存器(DX 或 EDX)中。请注意，即使被除数没有“高”部分，“D”系列寄存器也被清零，而且在 DIV 指令执行期间，这些“ D”系列寄存器不能用于保存独立的值。

请记住，当操作数为一个内存值时，必须在该操作数的前面加上一个类型说明符：BYTE、WORD 或 DWORD。

DIV 不在寄存器中遗留任何信息。但是请注意，在 DIV 操作后，OF、SF、ZF、AF、PF 和 CF 的值都变为不确定状态。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

INC：递增操作数指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*       * * * *     IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
INC r8
INC m8
INC r16
INC m16
INC r32     386+
INC m32     386+
```

举例

```
INC AL
INC BX
INC EDX
INC BYTE [BP]        ; 递增位于[BP]处的字节
INC WORD [BX]        ; 递增位于[BX]处的字
INC DWORD [ESI]      ; 递增位于[ESI]处的双字
```

说明

请记住，INC 不能递增段寄存器。所有的半寄存器(r8)的操作码的长度为 2 个字节，但是所有的 16 位寄存器(r16)的操作码的长度为 1 个字节。如果你能够将只有低半部分包含数据的整个 16 位寄存器递减 1，那么请使用这个 16 位寄存

器以便节省一个字节的操作码。INC 不影响进位标志。(这是很容易犯的错误。)如同所有处理内存的指令一样，内存数据形式必须与诸如 BYTE、WORD 和 DWORD 之类的大小说明符一起使用。NASM 不假设使用任何一种大小说明符！

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

INT：软件中断指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
    * *             IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
INT3            ; INT 3 的 NASM 专属简写方式
INT i8
```

举例

```
INT3            ; NASM 需要这个来产生一个 INT 3 指令
INT 10H
```

说明

INT 触发一个位于内存的前 1024 字节中的 256 个向量之一的软件中断。操作数指定了从 0～255 之间的哪个向量。当一个中断被调用时，标志寄存器被与返回地址一起压入堆栈。IF 标志被清零，这样可以防止再有中断(无论是硬件中断还是软件中断)被识别，直至 IF 再次被设置为止。TF 标志也被清零。

该指令的一种特殊使用形式允许通过一条单字节指令调用中断 3。通过将某条指令替换为用于调用中断 3 的单字节操作码，调试器可使用 Interrupt 3 在代码中设置断点。NASM 不识别这种形式，如果出于某种原因必须要使用 INT 3(那个指令形式没有多大用处，除非你正在编写一个调试器)，你必须使用一种特殊的

助记符形式，INT3，而不是 INT 3。这是更高级的内容，一定要小心。

几乎所有 INT 应用都会使用另外那种采用一个 8 位立即数的形式。

一般情况下，总是通过 IRET 指令从软件中断服务例程中返回。IRET 恢复被 INT 压入到堆栈中的标志，并且将 IF 置于清零状态，从而允许 CPU 继续响应其他的中断。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8 = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

IRET：从中断中返回

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	OF: 溢出标志	TF: 陷阱标志	AF: 辅助进位标志
*	*	*	*	*	*	*	*	*	DF: 方向标志	SF: 符号标志	PF: 奇偶标志
									IF: 中断标志	ZF: 零标志	CF: 进位标志

合法形式

```
IRET
```

举例

```
IRET
```

说明

IRET 必须被用于从通过 INT 或者中断硬件(如串行端口等)调用的中断服务例程中退出。IRET 从栈顶弹出返回地址到 CS 和 IP 中，然后再从堆栈中弹出下一个字到标志寄存器。所有标志都受到影响。

如果中断是由硬件引起的，在 IRET 执行之前，可能需要额外的步骤来准备硬件以便用于下一次中断。你可以参考相关硬件文档。

当使用 NASM 时，为 IRET 产生的实际操作码取决于 BITS 的设置，并且控制着是产生 16 位返回值还是 32 位返回值。

在保护模式下，用户空间应用程序不能包含中断服务例程，因此不能使用

IRET。

```
r8  = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr  = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8  = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8  = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

J?：有条件跳转

影响的标志

```
O D I T S Z A P C    OF: 溢出标志  TF: 陷阱标志  AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志  SF: 符号标志  PF: 奇偶标志
    <none>           IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式	描述	如果标志为……则跳转
JA/JNBE d	(如果以上则跳转/如果不以下或等于则跳转)	CF=0 并且 ZF=0
JAE/JNB d	(如果以上或等于则跳转/如果不以下则跳转)	CF=0
JB/JNAE d	(如果以下则跳转/如果不以上或等于则跳转)	CF=1
JBE/JNA d	(如果以下或等于则跳转/如果不以上则跳转)	CF=1 或者 ZF=1
JE/JZ d	(如果相等则跳转/如果为零则跳转)	ZF=1
JNE/JNZ d	(如果不相等则跳转/如果不为零则跳转)	ZF=0
JG/JNLE d	(如果大于则跳转/如果小于或等于则跳转)	ZF=0 并且 SF=OF
JGE/JNL d	(如果大于或等于则跳转/如果不小于则跳转)	SF=OF
JL/JNGE d	(如果小于则跳转/如果不大于等于则跳转)	SF≠OF
JLE/JNG d	(如果小于或者等于则跳转/如果不大于则跳转)	ZF=1 或者 SF≠OF
JC d	(如果进位标志得到设置，则跳转)	CF=1
JNC d	(如果进位标志没有得到设置，则跳转)	CF=0
JO d	(如果溢出标志得到设置，则跳转)	OF=1
JNO d	(如果溢出标志没有得到设置，则跳转)	OF=0
JP/JPE d	(如果奇偶标志位 PF 得到设置，则跳转/奇偶性为偶数时跳转)	PF=1

续表

合法形式	描述	如果标志为……则跳转
JNP/JPO d	(如果奇偶标志位 PF 没有得到设置，则跳转/奇偶性为奇数时跳转)	PF=0
JS d	(如果符号标志得到设置，则跳转)	SF=1
JNS d	(如果符号标志没有得到设置，则跳转)	SF=0

不带 NEAR 的 d = 8 位有符号位移；在 d 的前面使用 NEAR 以指定段宽度范围内的位移。

举例

```
JB HalfSplit        ; 如果 CF=1，则跳转
JLE TooLow          ; 如果 ZF=1 或者 SFOF，则跳转
JG NEAR WayOut      ; 在实模式下，如果比 16 位位移量大，则跳转
                    ; 或者在 32 位保护模式下，如果比 32 位位移量大，则跳转
```

说明

默认情况下，如果某个条件为真，所有这些指令都进行短跳转(往前 127 个字节或者往后 128 个字节)，如果条件不为真，则跳转失败。这些条件都涉及标志位，并且我们所说的标志条件已经在合法形式列右侧的助记符及其描述中给出。

那些包含“以上(above)”或者“以下(below)”的助记符用于无符号比较，而那些包含“小于(less)”或者“大于(greater)”的助记符用于有符号比较。“等于”和“零”用于有符号比较和无符号比较皆可。

NASM 通过在指令助记符后面插入 NEAR 关键字允许我们使用段内宽度形式。这使得我们在实模式下可以使用 16 位有符号位移量，并且在 32 位保护模式下可以使用 32 位有符号的位移量。仅 386 和更高级别的 CPU 支持对 NEAR 的使用。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

JCXZ：如果 CX = 0，则跳转

影响的标志

```
O D I T S Z A P C    OF: 溢出标志  TF: 陷阱标志  AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志  SF: 符号标志  PF: 奇偶标志
      <none>         IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式

```
JCXZ <短位移>
```

举例

```
JCXZ AllDone   ;标号 AllDone 必须在 127 个字节以内!
```

说明

许多指令使用 CX 作为计数寄存器，JCXZ 允许你测试 CX，并且当看到 CX 已变为 0 时进行跳转。该跳转可能只是一个短跳转(即向前不超过 127 个字节或向后不超过 128 个字节)，并且如果当指令执行时 CX=0，该跳转被执行。如果 CX 为非 0 值时，执行失败，直接进入下一条指令。参见条件转移指令。

当使用 LOOP 指令时，我们经常使用 JCXZ 来绕过 CX = 0 的条件。由于在测试 CX= 0 之前，LOOP 指令递减 CX 中的值，如果你输入一个由 CX=0 来控制的循环，你将最终迭代该循环 65536 次，因此请使用 JCXZ。

```
r8  = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr  = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8  = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据              i8  = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8  = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

JECXZ：如果 ECX = 0，则跳转

影响的标志

```
O D I T S Z A P C    OF: 溢出标志  TF: 陷阱标志  AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志  SF: 符号标志  PF: 奇偶标志
      <none>         IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式

```
JECXZ <短位移>   386+
```

举例

```
JECXZ AllDone    ;标号 AllDone 必须在 127 个字节以内!
```

说明

本指令的操作与 JCXZ 相同，只不过寄存器测试的是 ECX，而不是 CX。

当使用 LOOP 指令时，我们经常使用 JECXZ 来绕过 ECX = 0 这一条件。由于在测试 ECX= 0 之前，LOOP 指令递减 ECX 中的值，所以如果你输入一个被 ECX=0 控制的循环，你将最终迭代该循环 2 147 483 648 次，因此，请使用 JECXZ。

```
r8 = AL AH BL BH CL CH DL DH r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS       r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据             m16 = 16 位内存数据
m32 = 32 位内存数据           i8 = 8 位立即数
i16 = 16 位立即数             i32 = 32 位立即数
d8 = 8 位有符号位移           d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

JMP：无条件转移指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>         IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
JMP <短位移>
JMP <近标号>
JMP <远标号>
JMP r16
JMP r32     386+
JMP m16
JMP m32     386+
```

举例

```
JMP RightCloseBy       ; 加或者减 128 个字节
JMP InsideMySegment    ; 跳转到一个距离 CS 偏移量为 16 位的地址处
JMP OutsideMySegment   ; 跳转到一个 32 位立即地址处
JMP DX                 ; 跳转到一个存放在 DX 寄存器中的 16 位偏移量处
JMP EAX                ; 跳转到一个存放在 EAX 寄存器中的 32 位偏移量处
JMP WORD [BX+DI+17]    ; 跳转到存放在[BX+DI+17]中的近地址处
JMP DWORD [EBX+EDI+17] ; 跳转到一个存放在[EBX+EDI+17]中的完整 32 位地址处
```

说明

JMP 无条件地将控制权移交到一个由单操作数给出的目标处。除了已定义的标号外，JMP 还可以将控制权移交到一个距离 IP 的偏移量为存放在某个通用寄存器中的 16 位有符号值的地址处(或距离 EIP 的偏移量为 32 位有符号值的地址处)，或者移交到一个存放在内存中的地址处(或为近地址，或为远地址)，然后通过任何合法的寻址方式进行访问。当我们在内存中创建跳转表时(注：在内存中跳转表是一个地址数组)，可以使用 m16 和 m32 的形式。例如，JMP [BX+DI+17]将控制权转移到代码段中某个 16 位的偏移量处，该偏移量可从基址索引位移地址[BX+DI+17] 中找到。

没有任何标记受到影响，并且没有任何返回地址被压入堆栈，这一点与 CALL 指令有所不同。

```
r8 = AL AH BL BH CL CH DL DH  r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS        r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            16 = 16 位有符号位移
d32 = 32 位无符号位移
```

LEA：加载有效地址

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>        IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
LEA r16,m<任意大小>
LEA r32,m<任意大小>   386+
```

举例

```
LEA EBX,[EAX+EDX*4+128]      ;将计算出来的地址加载到 EBX 中
LEA BP,MyWordVar             ;将 MyWordVar 偏移地址加载到 BP 中
```

说明

LEA 导出源操作数距离它所在的段的起始位置的偏移地址，并将其加载到目标操作数中。目标操作数必须是一个寄存器，不能是内存。源操作数必须是一个内存操作数，但是可以为任意大小。存储在目标操作数中的地址是内存中的源操作数的第一个字节的地址，源操作数在内存中的大小并不重要。

这是一种在一个程序或中断调用之前，将一个变量的地址放入某个寄存器中的又好又干净的方式。

LEA 也可用于执行寄存器数学运算，因为在第二个操作数中指定的地址是计算出来的，而不是通过直接访问得来的。因此，该地址可以是你的程序无权访问的地址。任何可被表示为一个有效地址计算的数学运算都可以通过 LEA 来执行。

这是 NASM 的为数不多的在一个用于给出某个内存位置的操作数前面不需要大小标识符的地方——再次重申一下，因为 LEA 只是计算地址，却不往那个地址移动数据，或者从那个地址移出数据。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8 = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

LOOP：循环，直到 CX/ECX = 0

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>         IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
LOOP d8
```

举例

```
LOOP PokeValue
```

说明

LOOP 指令是一个递减计数器指令、测试指令和转移指令的组合。它在 16 位模式下使用 CX 寄存器，在 32 位模式下使用 ECX 寄存器。LOOP 指令在两种模式下的操作逻辑是相同的，这里，我以 32 位编码为例。

LOOP 指令同时可作为一条 DEC ECX 指令、一条 CMP ECX，0 指令、一条 JZ 指令而执行，因此可以简化代码。表示循环重复次数的计数器必须被初始化为加载到 ECX 中。当 LOOP 指令执行时，它首先递减 ECX。然后测试 ECX 是否等于 0。如果 ECX 不为 0，LOOP 将控制转移到其操作数指定的位移处：

```
MOV ECX,17
DoIt: CALL CrunchIt
CALL StuffIt
LOOP DoIt
```

这里将两个过程调用重复执行 17 次。前 16 次之后，ECX 中仍然为非零值，LOOP 将控制转移到 DoIt。然而，在第 17 次循环中，LOOP 将 ECX 递减为 0，然后当它测试 CX 时，执行失败，程序流程进入它后面的下一个按顺序出场的指令中。

LOOP 不改变任何标志，即使当 ECX 已经递减为 0 时。警告：一定要留神初始条件！如果在 16 位模式下，CX 的初始值为 0，LOOP 将递减它至 65 535(0FFFFH)，然后执行循环 65 535 次。更糟糕的是，如果你工作在 32 位保护模式下，进入一个 ECX = 0 的循环，则循环将会被执行超过 2 亿次，这可能需要很长的时间，足以看起来像一个系统死锁。

```
r8 = AL AH BL BH CL CH DL DH     r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS           r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8 = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

LOOPNZ/LOOPNE：当 CX/ECX>0 且 ZF=0 时循环

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
        <none>       IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
LOOPNZ d8
LOOPNE d8
```

举例

```
LOOPNZ StartProcess
LOOPNE GoSomewhere
```

说明

LOOPNZ 和 LOOPNE 是同义词，并且产生相同的操作码。像 LOOP 一样，它们或者使用 CX，或者使用 ECX，取决于 CPU 的 BITS 的设置的不同，也就是说，取决于 CPU 的当前“位”。LOOPNZ / LOOPNE 递减 ECX，并且如果 ECX 不为 0，且零标志 ZF 为零，跳转到目标操作数指定的位置。否则，执行进入下一条指令。

这意味着该循环主要被 ZF 控制。如果 ZF 依然为 0，那么循环将继续，一直到 ECX 递减为 0，但是只要 ZF 一被设置为 1，循环就马上终止。我们可以把它看成是“当零标志为零时则循环”。

请记住，LOOPNZ 本身并不影响 ZF。在循环内部，必须有些指令(通常为字符串指令)在 CX/ ECX 被倒计数为 0 之前，做某些事情来影响 ZF，以便让循环终止。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

LOOPZ/LOOPE：当 CX/ECX>0 且 ZF=1 时循环

影响的标志

```
O  D  I  T  S  Z  A  P  C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F  F  F  F  F  F  F  F  F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
         <none>              IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
LOOPZ d8
LOOPE d8
```

举例

```
LOOPZ SenseOneShots
LOOPE CRCGenerate
```

说明

LOOPZ 和 LOOPE 是同义词，并且产生相同的操作码。像 LOOP 一样，它们或者使用 CX，或者使用 ECX，取决于 CPU 的 BITS 的设置的不同，也就是说，取决于 CPU 的当前“位”。LOOPZ/LOOPE 递减 ECX，并且如果 ECX 不为 0 且零标志 ZF 为 1 时跳转到目标操作数指定的位置。否则，执行进入下一条指令。

这意味着该循环主要由 ZF 控制。如果 ZF 依然为 1，那么循环将继续，一直到 CX 被递减为 0，但是只要 ZF 一被清零，循环就马上终止。我们可以把它看成是“当零标志为 1 时则循环”。

请记住，LOOPZ 本身并不影响 ZF。在循环内部，必须有些指令(通常为字符串指令)在 CX/ ECX 被倒计数为 0 之前，做某些事情来影响 ZF，以便让循环终止。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

MOV：移动(复制)右边的操作数到左边的操作数中

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	OF：溢出标志	TF：陷阱标志	AF：辅助进位标志
				<none>					DF：方向标志	SF：符号标志	PF：奇偶标志
									IF：中断标志	ZF：零标志	CF：进位标志

合法形式

```
MOV r8,r8
MOV m8,r8
MOV r8,m8
MOV r8,i8
MOV m8,i8
MOV r16,r16
MOV m16,r16
MOV r16,m16
MOV m16,i16
MOV r16,i16
MOV r32,r32   386+
MOV m32,r32   386+
MOV r32,m32   386+
MOV r32,i32   386+
MOV m32,i32   386+
MOV sr,r16
MOV sr,m16
MOV r16,sr
MOV m16,sr
```

举例

```
MOV AL,BH
MOV EBX,EDI
MOV BP,ES
MOV ES,AX
MOV AX,0B800H
MOV ES:[BX],0FFFFH
MOV CX,[SI+Inset]
```

说明

这也许是最常用的指令。源(右)操作数被复制到左边(目标)的操作数中。源操作数不变。标志位不受影响。

```
r8 = AL AH BL BH CL CH DL DH  r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS        r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

MOVS：移动字符串指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>         IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
MOVSB
MOVSW
MOVSD  386+
```

举例

```
MOVSB          ; 将[ESI] 处的字节复制到[EDI]中
MOVSW          ; 将[ESI] 处的字复制到[EDI]中
MOVSD          ; 将[ESI] 处的双字复制到[EDI]中
REP MOVSB      ; 将从[ESI]处开始的内存区域复制到从[EDI]处开始的内存区域，
               ; 重复 CX/ECX 次，每次复制一个字节
```

说明

MOVS 以 8 位(MOVSB)、16 位(MOVSW)或者 32 位(MOVSD)块为单位，将数据从 ESI 中的地址指定的内存中复制到 EDI 中的地址指定的内存中。

对于 16 位模式而言，ES 必须是目标操作数的段地址，并且不能被改写。在 32 位保护模式下，所有的短地址都是一致的；因此，ES 并不需要被显式地指

定。类似的，ES:DI 或者 EDI 必须一直为目标操作数的偏移地址。

通过将一个操作重复计数值(不是数据单元计数值！)放入 CX/ ECX 中，并且在助记符前面加上 rep 前缀，MOVS 可以像自动“机枪”一样，自动地将数据从[ESI]处开始的内存中复制到从[EDI]处开始的内存中。

每次复制操作之后，SI/ESI 和 DI/EDI 被调整了(见下段)或者 1(对于 8 位操作而言)，或者 2(对于 16 位操作而言)，或者 4(对于 32 位操作而言)，同时 CX/ECX 被递减 1。不要忘记，CX/ECX 对操作次数进行计数(有多少次单个数据项被从源地址复制到目的地)，而不是对字节数进行计数！

如果方向标志被(CLD 指令)清零，调整意味着递增；如果方向标志已被(STD 指令)设置为 1，调整意味着递减。

```
r8  = AL AH BL BH CL CH DL DH  r16 = AX BX CX DX BP SP SI DI
sr  = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8  = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据             i8  = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8  = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

MOVSX：带符号扩展移动(复制)指令

影响的标志

```
O  D  I  T  S  Z  A  P  C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F  F  F  F  F  F  F  F  F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
         <none>             IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
MOVSX r16,r8      386+
MOVSX r16,m8      386+
MOVSX r32,r8      386+
MOVSX r32,m8      386+
MOVSX r32,r16     386+
MOVSX r32,m16     386+
```

举例

```
MOVSX AX,AL
MOVSX CX,BYTE  [EDI]          ; 对位于[ESI] 处的字节采取行动
```

```
MOVSX ECX,DL
MOVSX ESI,DWORD [EBX+EDI]    ;将位于[EBX+EDI]处的双字采取行动
```

说明

MOVSX 的操作与 MOV 类似，但是采用带符号扩展的方式将值从源操作数复制到目标操作数中。也就是说，它将较小的源操作数的符号位携带到较大的目标操作数中。例如，采用这种方式，当将 AX 中的 16 位有符号值复制到 32 位寄存器 EDX 中时，该值仍将是一个有符号值。如果采用不带符号的扩展， 那么 AX 中的有符号值的符号位将被当成一个普通的二进制位而复制到 EDX 中，并且 EDX 中的值将会容忍原本该被认为相同的值居然没有任何相似之处。

目标操作数必须是一个寄存器。MOVSX 可以从一个内存位置复制数据，但是不能将数据复制到一个存储位置。还要注意，目标操作数必须是一个比源操作数位数更大的值；也就是说，MOVSX 能够将一个 8 位或 16 位的值复制到一个 32 位的值中，但是不能将一个 16 位值复制到 16 位值中，也不能将一个 32 位值复制到 32 位值中。

MOVSX 只出现在 386 及更高级别的 CPU 上。它不影响任何标志位。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

MUL：无符号整数除法指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*         ? ? ? ? * IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
MUL r8
MUL m8
MUL r16
MUL m16
```

```
MUL r32    386+
MUL m32    386+
```

举例

```
MUL CH              ; AL * CH --> AX
MUL BX              ; AX * BX --> DX:AX
MUL ECX             ; EAX * ECX --> EDX:EAX
MUL WORD [BX+DI]    ; AX * [BX+DI] --> DX:AX
```

说明

MUL 将它的操作数与 AL 或者 AX 或者 EAX 相乘，并将结果存放到 AX 或者 DX:AX 或者 EDX:EAX 中。如果 MUL 的操作数为 8 位(或者是一个 8 位寄存器，或者是一个 8 位内存操作数)，其结果将被存放到 AX 中。这意味着，即使结果能够完全存放在 AL 中，AH 也将受到影响。

同样，如果 MUL 的操作数为 16 位，结果将被存放在 DX：AX 中，即使整个结果完全能够在 AX 中存放！人们很容易忘记，在 16 位乘法中，MUL 影响 DX，在 32 位乘法中，MUL 影响 EDX。记住这一点！

注意：人们很容易假设 IMUL 与 MUL 相同，除了 IMUL 能够操作有符号值之外。事实并非如此：IMUL 有更多的合法指令形式，而且其复杂性大大超过 MUL。

如果计算的结果为 0，进位和溢出标志被清零；否则，两者都被设置为 1。请记住，SF、ZF、 AF 和 PF 在 MUL 执行后，都变为不确定状态。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

NEG：求补(2 的补码；即，与−1 相乘）指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*       * * * * *    IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
NEG r8
NEG m8
NEG r16
NEG m16
NEG r32    386+
NEG m32    386+
```

举例

```
NEG AL
NEG DX
NEG ECX
NEG BYTE [BX]      ; 将位于[BX]处的字节量求补
NEG WORD [DI]      ; 将位于[BX]处的长度为一个字的量求补
NEG DWORD [EAX]    ; 将位于[EAX]处的长度为一个双字的量求补
```

说明

这是将一个值与-1 相乘的汇编语言等值。请记住，求补并不是简单地将操作数中的每个位取反。(另一个指令，NOT，是这样做的。）这个过程也被叫作生成一个值的 2 的补码。一个值的 2 的补码与该值相加，产生的结果为零。-1=$FF；-2=$FE；-3=$FD，等等。

如果操作数为 0，CF 被清零且 ZF 被设置，否则，CF 被设置且 ZF 被清零。如果操作数包含操作数位数范围内的最大负值，则操作数不变，但 OF 和 CF 被设置。如果结果为负数，SF 被设置，否则 SF 被清零。如果结果的低 8 位包含偶数个设置为 1 的位，PF 被设置，否则，PF 被清零。

注意：对于内存数据，我们必须使用大小说明符(BYTE、WORD、DWORD)!

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

NOP：空操作指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
        <none>      IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
NOP
```

举例

```
NOP
```

说明

这是所有 86 家族机器指令中最容易理解的，它什么都不做。它的工作就是在指令序列中占用一个空间。当被 CPU 取出时，NOP 被认为执行 XCHG AX,AX。因此，它其实是做了一些工作，但这些工作不是有用的工作，并没有改变任何数据，标志也不受影响。NOP 用于在调试过程中使一条机器指令“执行空操作”，以便为将来的过程调用、中断调用或延时循环预留空间——虽然鉴于现代 x86 CPU 的能力，在对执行代码进行各种上下文敏感的优化时，延时循环已经不再可能量化。在编译阶段精确地预测执行一条指令所需花费的时间已经再无可能！

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

NOT：逻辑非(1 的补码)指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF:陷阱标志   AF: 辅助进位标志
```

```
F  F  F  F  F  F  F  F  F   DF: 方向标志  SF: 符号标志  PF: 奇偶标志
          <none>            IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式

```
NOT r8
NOT m8
NOT r16
NOT m16
NOT r32    386+
NOT m32    386+
```

举例

```
NOT CL
NOT DX
NOT EBX
NOT WORD [SI+5]
```

说明

NOT 指令把操作数中的每一位都分别取反。也就是说，每一个值为 1 的位都变为 0，每一个值为 0 的位都变为 1。这是“逻辑非”或“1 的补码”操作。如果想执行求补或者求 2 的补码操作，参见 NEG 指令。

在执行 NOT 指令后，值 FFH 将变为 0；值 AAH 将变为 55H。请注意，零标志不受影响，即使 NOT 迫使其操作数变为 0。

注意：对于内存数据，我们必须使用大小说明符(BYTE、WORD、DWORD)!

```
r8  = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr  = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8  = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据              i8  = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8  = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

OR：逻辑或指令

影响的标志

```
O  D  I  T  S  Z  A  P  C   OF: 溢出标志  TF:陷阱标志  AF: 辅助进位标志
```

```
F  F  F  F  F  F  F  F  F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*        *  *  ?  *  *       IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
OR r8,r8
OR m8,r8
OR r8,m8
OR r16,r16
OR m16,r16
OR r16,m16
OR r32,r32          386+
OR m32,r32          386+
OR r32,m32          386+
OR r8,i8
OR m8,i8
OR r16,i16
OR m16,i16
OR r32,i32          386+
OR m32,i32          386+
OR AL,i8
OR AX,i16
OR EAX,i32          386+
```

举例

```
OR EBX,EDI
OR AX,0FFFFH    ;使用单字节操作码
OR AL,42H       ;使用单字节操作码
OR WORD [EBX],0B8C0H
OR WORD [BP+SI],DX
```

说明

OR 指令对它的两个操作数执行逻辑或运算。一旦操作完成，结果将替换目标操作数。OR 按位执行，源操作数的第 0 位和目标操作数位的第 0 位相或，源操作数的第 1 位和目标操作数位的第 1 位相或，等等。如果有一个操作数为 1，该操作将产生一个 1，只有当两个操作数都为 0 时才产生 0。请注意，该操作使辅助进标志变为不确定状态。CF 和 OF 被清 0，其他受影响的标志位根据操作的结果进行设置。

注意：对于内存数据，我们必须使用大小说明符(BYTE、WORD、DWORD)!

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

POP：将栈顶元素弹出到操作数中

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>         IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
POP r16
POP m16
POP r32
POP m32
POP sr
```

举例

```
POP WORD [BX]
POP EAX
POP DX
POP DWORD [EAX+ECX]
POP ES
```

说明

从堆栈中弹出一个 8 位的项是不可能的。还要记住，栈顶元素被定义(在 16 位模式下)为一个位于 SS:SP 处的字，而且没有办法通过前缀 SS 以寻址方式来重写。在 32 位模式下，栈顶元素是位于[ESP]处的双字值。有一对儿单独的指令，PUSHF 和 POPF，可用于压入和弹出标志寄存器。

所有寄存器形式的使用都拥有单字节操作码。NASM 识别它们并自动生成它们，尽管在 CPU 的指令解码逻辑中，有更大形式的操作码。

```
r8 = AL AH BL BH CL CH DL DH      r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS            r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                  m16 = 16 位内存数据
m32 = 32 位内存数据                i8 = 8 位立即数
i16 = 16 位立即数                  i32 = 32 位立即数
d8 = 8 位有符号位移                d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

POPA/POPAD：弹出所有的通用寄存器

影响的标志

O F	D F	I F	T F	S F	Z F	A F	P F	C F
				<none>				

OF：溢出标志　TF：陷阱标志　AF：辅助进位标志
DF：方向标志　SF：符号标志　PF：奇偶标志
IF：中断标志　ZF：零标志　CF：进位标志

合法形式

```
POPA 286+
POPAD 386+
```

举例

```
POPA    ; 从堆栈中弹出所有的 8 个 16 位寄存器，同时丢弃 SP 的值
POPAD   ; 从堆栈中弹出所有的 8 个 32 位寄存器，同时丢弃 ESP 的值
```

说明

POPA 将所有的 16 位通用寄存器弹出堆栈。本指令出现在 286 及更高级别的 CPU 上，在 8086/8088 上不可用。这 8 个 16 位通用寄存器按照如下顺序弹出：

```
DI, SI, BP, SP, BX, DX, CX, AX
```

这里有一个革新：被 POPA 从堆栈中弹出的 SP 的值并不被弹回到 SP 中！(这将是疯狂的，因为当从堆栈中弹出值的时候，我们正在使用 SP 来管理堆栈。) 当指令执行到达它时，位于堆栈上的 SP 处的值只是被简单地丢弃。

POPAD 是 POPA 的 32 位版本。它按照如下顺序从堆栈中弹出所有的 8 个 32 位寄存器的值到相应的寄存器中，ESP 除外：

```
EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX
```

ESP 压入堆栈的值被 POPAD 丢弃掉，而不是替换 ESP 的当前值。

POPA 和 POPAD 通常用于与 PUSHA 和 PUSHAD 相呼应，但是没有任何东西承诺这一点。如果你从堆栈中弹出垃圾值到通用寄存器中，那么有趣的事情(有点像古老的中国式诅咒)能够而且很可能会发生。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8 = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

POPF：将栈顶值弹出到 16 位标志寄存器中

影响的标志

```
O D I T S Z A P C   OF: 溢出标志  TF: 陷阱标志  AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志  SF: 符号标志  PF: 奇偶标志
* * * * * * * * *   IF: 中断标志  ZF: 零标志    CF: 进位标志
```

合法形式

```
POPF
```

举例

```
POPF
```

说明

POPF 弹出堆栈顶部的 16 位字到标志寄存器中。堆栈顶部的元素被定义为位于 SS:SP 处的字，而且该值无法通过前缀来进行重写。

当这个字从堆栈中弹出后，SP 被递增 2。请记住，SP 始终或者指向一个空堆栈或者指向真实的数据。有一对单独的指令，PUSH 和 POP，可用于压入和弹出其他的寄存器和内存数据。

PUSHF 和 POPF 最常用于编写 16 位中断服务程序，在那里你必须能够保存和恢复环境——也就是说，所有的机器寄存器——以避免在服务于这个中断期间干扰机器的操作。

有一对儿单独的指令，PUSHFD 和 POPFD，可以压入和弹出 32 位 Eflags 寄存器。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

POPFD：将栈顶元素弹出到 EFlags 寄存器中

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
* * * * * * * * *   IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
POPFD
```

举例

```
POPFD  386+
```

说明

POPFD 从堆栈的顶部弹出双字(4 字节)到 EFlags 寄存器中。在 32 位保护模式下，栈顶被定义为位于[ESP]处的双字。

当栈顶的字被弹出后，ESP 被递增 4。请记住，ESP 始终或者指向一个空堆栈或者指向真实的数据。有一对单独的指令，PUSH 和 POP，用于压入和弹出其他的寄存器和内存数据，16 位和 32 位均可。还有一对单独的指令，PUSHF 和 POPF，用于压入和弹出 16 位标志寄存器。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

PUSH：将操作数压入栈顶指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
        <none>       IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
PUSH r16
PUSH m16
PUSH r32      386+
PUSH m32      386+
PUSH sr
PUSH i8       286+
PUSH i16      286+
PUSH i32      386+
```

举例

```
PUSH WORD [EBX]
PUSH EAX
PUSH DI
PUSH DWORD 5
PUSH WORD 1000H
```

说明

将一个 8 位的项压入堆栈是不可能的。同时要记住，栈顶被定义(在 16 位模式下)为位于 SS:SP 地址处的字，并且该值无法使用前缀 SS 以寻址方式重写。在 32 位模式下，栈顶是位于[ESP]处的双字值。有一对单独的指令，PUSHF 和 POPF，可用于压入和弹出标志寄存器。

PUSH 有一种特殊的形式，可实现将段寄存器压入堆栈，但是那些没有在这里列出，因为它们不能在普通的 Linux 用户空间编程中使用。

也要记住，在压栈发生之前 SP / ESP 被递减；SP 或者指向一个空堆栈或者指向真实的数据。

```
r8 = AL AH BL BH CL CH DL DH  r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS        r32 = EAX EBX ECX EDX EBP ESP ESI EDI
```

```
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

PUSHA：将所有的 16 位通用寄存器压入堆栈

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
        <none>      IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
PUSHA  286+
```

举例

```
PUSHA
```

说明

PUSHA 将所有的 16 位通用寄存器压入堆栈。这条指令出现在 286 及更高级别的 CPU 上，在 8086/8088 上不可用。

寄存器被按照如下的顺序压入堆栈：

```
AX, CX, DX, BX, SP, BP, SI, DI
```

但是，请注意，压入堆栈的 SP 的值是在第一个寄存器被压入堆栈之前 SP 所拥有的值。在执行 PUSHA 的过程中，堆栈指针递减 16 个字节(8*每个寄存器 2 个字节)。

压入堆栈的 SP 的值被认为是无用的，并且由 POPA 丢弃。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

PUSHAD：将所有的 32 位通用寄存器压入堆栈

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
          <none>    IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
PUSHAD     386+
```

举例

```
PUSHAD
```

说明

PUSHAD 将所有的 32 位通用寄存器压入堆栈。这条指令出现在 386 及更高级别的 CPU 上，在 8086/8088 或者 286 上不可用。

寄存器被按照如下的顺序压入堆栈：

```
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
```

但是，请注意，压入堆栈的 ESP 的值是在第一个寄存器被压入堆栈之前 ESP 所拥有的值。在执行 PUSHAD 的过程中，堆栈指针递减 32 个字节(8*每个寄存器 4 个字节)。

压入堆栈的 ESP 的值被认为是无用的，并且由 POPAD 丢弃。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

PUSHF：将 16 位标志寄存器压入堆栈

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	OF：溢出标志	TF：陷阱标志	AF：辅助进位标志
				<none>					DF：方向标志	SF：符号标志	PF：奇偶标志
									IF：中断标志	ZF：零标志	CF：进位标志

合法形式

```
PUSHF
```

举例

```
PUSHF
```

说明

PUSHF 只是将 16 位标志寄存器的当前内容(不是 32 位的 EFlags！)压入堆栈的顶部。在 16 位模式下，栈顶被定义为位于 SS:SP 处的字，并且该值无法使用前缀重写。

在字进入堆栈之前，SP 被递减。请记住，SP 始终或者指向一个空堆栈或者指向真实的数据。有一对单独的指令，PUSH 和 POP，可用于压入和弹出其他的寄存器数据和内存数据。

当将标志压入堆栈时，标志寄存器不受影响，只有当通过 POPF 将它们弹回的时候才受影响。

PUSHF 和 POPF 最常用于编写中断服务程序，在那里你必须能够保存和恢复环境——也就是说，所有的机器寄存器——以避免在服务于这个中断期间干扰机器的操作。

r8 = AL AH BL BH CL CH DL DH
r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS
r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据
m16 = 16 位内存数据
m32 = 32 位内存数据
i8 = 8 位立即数
i16 = 16 位立即数
i32 = 32 位立即数
d8 = 8 位有符号位移
d16 = 16 位有符号位移
d32 = 32 位无符号位移

PUSHFD：将 32 位 EFlags 寄存器压入堆栈

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	OF：溢出标志	TF：陷阱标志	AF：辅助进位标志
				<none>					DF：方向标志	SF：符号标志	PF：奇偶标志
									IF：中断标志	ZF：零标志	CF：进位标志

合法形式

```
PUSHFD  386+
```

举例

```
PUSHFD
```

说明

PUSHFD 只是将 32 位 EFlags 寄存器的当前值压入栈顶。在 32 位模式下，栈顶被定义为位于[ESP]处的双字。在 EFlags 双字进入堆栈之前，ESP 被递减。记住，ESP 始终或者指向一个空堆栈或者指向真实的数据。有一对单独的指令，即 PUSH 和 POP，可用于压入和弹出其他的寄存器数据和内存数据，并且在 286 及以后的处理器中，可以压入和弹出立即数。

当我们将标志压入堆栈的时候，EFlags 寄存器不受影响，只有当通过 POPFD 指令将它们弹回时 EFlags 寄存器才受影响。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

RET：从过程中返回

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	OF：溢出标志	TF：陷阱标志	AF：辅助进位标志
									DF：方向标志	SF：符号标志	PF：奇偶标志

<none>	IF: 中断标志	ZF: 零标志	CF: 进位标志

合法形式

```
RET
RETN
RETF
RET i8
RETN i8
RET i16
RETF i16
```

举例

```
RET
RET 12H
RETN
RETF 117H
```

说明

返回有两种类型：Near 和 Far，Near 指的是返回到当前的代码段范围内，Far 指的是返回到一些其他的代码段中。(这在 32 位保护模式下不会带来什么问题，因为在这种模式下只有一个代码段。)通常情况下，我们使用 RET 形式，汇编编译器将其解析为一个 Near 或 FAR 的返回操作码，以匹配过程定义中使用的 NEAR 或者 FAR 说明符。我们可以在必要的时候，指定 RETF 或 RETN。

RET 可以采用一个操作数来指明当从过程中返回时，需要释放多少个字节的堆栈空间。从堆栈指针中减去这个数可以擦除在过程调用之前压入堆栈的、用于该过程使用的数据项。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8 = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

ROL：循环左移指令

影响的标志

```
O  D  I  T  S  Z  A  P  C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F  F  F  F  F  F  F  F  F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*                       *   IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
ROL r8,1
ROL m8,1
ROL r16,1
ROL m16,1
ROL r32,1      386+
ROL m32,1      386+
ROL r8,CL
ROL m8,CL
ROL r16,CL
ROL m16,CL
ROL r32,CL     386+
ROL m32,CL     386+
ROL r8,i8      286+
ROL m8,i8      286+
ROL r16,i8     286+
ROL m16,i8     286+
ROL r32,i8     386+
ROL m32,i8     386+
```

举例

```
ROL AL,1
ROL WORD [BX+SI],CL
ROL BP,1
ROL DWORD [EBX+ESI],9
ROL BP,CL
```

说明

ROL 把目标操作数中的位循环左移，其中左边是指面向最高有效位(MSB)的方向。一次循环就是一次环绕移位(可以与 SHL 和 SHR 指令进行对比)：操作数最左侧的位转移到最右侧的位，所有中间位往左侧移动 1 位。除了移位操作采

用的方向不同之外，ROL 与 ROR 相同。

被移动的位的位置编号可以或者是一个 8 位立即数，或者由 CL 中而非 CX 或 ECX 中的值来指定。(8086 和 8088 被限制为立即数 1。)请注意，虽然 CL 可以接受一个最大可达 255 的值，但是将一个值移动超过任何大于 16 的位数是没有意义的，即使该移位操作实际上是在 8086 和 8088 上完成的。(除了运行在虚拟 86 模式下之外，286 以及后来的处理器限制了对于本机字长的操作数进行移位的位数。)

在每次移位操作中，最左边的位被复制到进位标志中。OF 只被一次移动一位形式的 ROL 所修改；在通过 CL 移动形式的 ROL 指令执行完毕后，OF 变为不确定状态。

```
r8 = AL AH BL BH CL CH DL DH    r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                m16 = 16 位内存数据
m32 = 32 位内存数据              i8 = 8 位立即数
i16 = 16 位立即数                i32 = 32 位立即数
d8 = 8 位有符号位移              d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

ROR：循环右移指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志    TF: 陷阱标志    AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志    SF: 符号标志    PF: 奇偶标志
*               *    IF: 中断标志    ZF: 零标志      CF: 进位标志
```

合法形式

```
ROR r8,1
ROR m8,1
ROR r16,1
ROR m16,1
ROR r32,1         386+
ROR m32,1         386+
ROR r8,CL
ROR m8,CL
ROR r16,CL
ROR m16,CL
ROR r32,CL        386+
```

```
ROR m32,CL         386+
ROR r8,i8          286+
ROR m8,i8          286+
ROR r16,i8         286+
ROR m16,i8         286+
ROR r32,i8         386+
ROR m32,i8         386+
```

举例

```
ROR AL,1
ROR WORD [BX+SI],CL
ROR BP,1
ROR DWORD [EBX+ESI],9
ROR BP,CL
```

说明

ROR 把目标操作数中的位循环右移，其中右边是指面向最低有效位(LSB)的方向。一次循环是就是一次环绕移位(可以与 SHL 和 SHR 指令进行对比)：操作数最右侧的位转移到最左侧的位，所有中间位往右侧移动 1 位。除了移位操作采用的方向不同之外，ROR 与 ROL 相同。

被移动的位的位置编号可以或者是一个 8 位立即数，或者由 CL 中而非 CX 或 ECX 中的值来指定。(8086 和 8088 被限制为立即数 1。)请注意，虽然 CL 可以接受一个最大可达 255 的值，但是将一个值移动超过任何大于 16 的位数是没有意义的，即使该移位操作实际上是在 8086 和 8088 上完成的。(除了运行在虚拟 86 模式下之外，286 以及后来的处理器限制了对于本机字长的操作数进行移位的位数。)

在每次移位操作中，最右边的位被复制到进位标志中。OF 只被一次移动一位形式的 ROR 所修改；在通过 CL 移动形式的 ROR 指令执行完毕后，OF 变为不确定状态。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

SBB：带借位减法指令

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F			
*				*	*	*	*	*			

OF：溢出标志　TF：陷阱标志　AF：辅助进位标志
DF：方向标志　SF：符号标志　PF：奇偶标志
IF：中断标志　ZF：零标志　CF：进位标志

合法形式

```
SBB r8,r8
SBB m8,r8
SBB r8,m8
SBB r16,r16
SBB m16,r16
SBB r16,m16
SBB r32,r32         386+
SBB m32,r32         386+
SBB r32,m32         386+
SBB r8,i8
SBB m8,i8
SBB r16,i16
SBB m16,i16
SBB r32,i32         386+
SBB m32,i32         386+
SBB r16,i8
SBB m16,i8
SBB r32,i8          386+
SBB m32,i8          386+
SBB AL,i8
SBB AX,i16
SBB EAX,i32         386+
```

举例

```
SBB BX,DI
SBB AX,0FFFFH       ;使用单字节操作码
SBB AL,42H          ;使用单字节操作码
SBB BP,17H
SBB WORD [BX+SI+Inset],5
SBB WORD [ES:BX],0B800H
```

说明

SBB 执行带借位的减法，其中源操作数以及进位标志被从目标操作数中减掉，然后用结果替换目标操作数。如果结果为负，进位标志将被设置。如果想执行不考虑进位标志(即没有借位)的减法，请使用 SUB 指令。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

SHL：逻辑左移指令

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	DF：方向标志	SF：符号标志	PF：奇偶标志
*				*	*	?	*	*	IF：中断标志	ZF：零标志	CF：进位标志

OF：溢出标志　TF：陷阱标志　AF：辅助进位标志

合法形式

```
SHL r8,1
SHL m8,1
SHL r16,1
SHL m16,1
SHL r32,1        386+
SHL m32,1        386+
SHL r8,CL
SHL m8,CL
SHL r16,CL
SHL m16,CL
SHL r32,CL       386+
SHL m32,CL       386+
SHL r8,i8        286+
SHL m8,i8        286+
SHL r16,i8       286+
SHL m16,i8       286+
SHL r32,i8       386+
SHL m32,i8       386+
```

举例

```
SHL AL,1
SHL WORD [BX+SI],CL
SHL BP,1
SHL EAX,9
SHL BP,CL
```

说明

SHL 将目标操作数中的位左移，其中左边是指面向最高有效位(MSB)的方向。被移动的位的位置编号可以或者是一个 8 位的立即数，或者由 CL 中而非 CX 或 ECX 中的值来指定。(8086 和 8088 被限制为立即数 1。)请注意，虽然 CL 可以接受一个最大可达 255 的值，但是将一个值移动任何大于 16 的位数或者在 32 位模式下任何大于 32 的位数是没有意义的，即使该移位操作实际上是在 8086 和 8088 上完成的。(除了运行在虚拟 86 模式下之外，286 以及后来的处理器限制了对于本机字长的操作数进行移位的位数。)

操作数的最左边的位被移到进位标志中；最右边的位被清除为 0。在本指令执行之后，辅助进位标志 AF 变为不确定状态。OF 只能被移动一位形式的该指令所修改；每当根据 CL 中的值进行移位的该指令形式执行后，OF 变为不确定状态。

SHL 是 SAL(算数左移指令)的同义词。除了移位操作的方向不同之外，SHL 与 SHR 相同。

```
r8 = AL AH BL BH CL CH DL DH     r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS            r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据                  m16 = 16 位内存数据
m32 = 32 位内存数据                i8 = 8 位立即数
i16 = 16 位立即数                  i32 = 32 位立即数
d8 = 8 位有符号位移                d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

SHR：逻辑右移指令

影响的标志

```
O D I T S Z A P C    OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F    DF: 方向标志   SF: 符号标志   PF: 奇偶标志
*       * * ? * *    IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
SHR r8,1
SHR m8,1
SHR r16,1
SHR m16,1
SHR r32,1          386+
SHR m32,1          386+
SHR r8,CL
SHR m8,CL
SHR r16,CL
SHR m16,CL
SHR r32,CL         386+
SHR m32,CL         386+
SHR r8,i8          286+
SHR m8,i8          286+
SHR r16,i8         286+
SHR m16,i8         286+
SHR r32,i8         386+
SHR m32,i8         386+
```

举例

```
SHR AL,1
SHR WORD [BX+SI],CL
SHR BP,1
SHR EAX,9
SHR BP,CL
```

说明

SHR 将目标操作数中的位右移，其中右边是指面向最低有效位(LSB)的方向。被移动的位的位置编号可以或者是一个 8 位立即数，或者由 CL 中而非 CX 或 ECX 中的值来指定。(8086 和 8088 被限制为立即数 1。)请注意，虽然 CL 可以接受一个最大可达 255 的值，但是将一个值移动任何大于 16 的位数或者在 32 位模式下任何大于 32 的位数是没有意义的，即使该移位操作实际上是在 8086 和 8088 上完成的。(除了运行在虚拟 86 模式下之外，286 以及后来的处理器限制了对于本机字长的操作数进行移位的位数。)

操作数的最右边的位被移到进位标志中；最左边的位被清除为 0。在本指令执行之后，辅助进位标志 AF 变为不确定状态。OF 只能由移动一位形式的该指令所修改；每当根据 CL 中的值移位的该指令形式执行后，OF 变为不确定状态。

SHR 是 SAR(算数右移指令)的同义词。除了移位操作的方向不同之外，SHR 与 SHL 相同。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

STC：设置进位标志(CF)

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
                *   IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
STC
```

举例

```
STC
```

说明

STC 将进位标志 CF 变为一个已知的设置状态(1)。当遇到一些在执行之前需要将进位标志位设置为 1 的任务时，我们可以使用它。它也可以用于从一个过程返回之前，将 CF 置于一个已知状态，以根据需要表明该过程是执行成功还是失败。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据              m16 = 16 位内存数据
m32 = 32 位内存数据            i8 = 8 位立即数
i16 = 16 位立即数              i32 = 32 位立即数
d8 = 8 位有符号位移            d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

STD：设置方向标志位(DF)

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
  *                 IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
STD
```

举例

```
STD
```

说明

STD 指令将方向标志 DF 的状态变为设置(1)状态。这会影响由重复字符串指令(如 STOS，SCAS 和 MOVS)所执行的调整。通常情况下，当 DF= 0 时，目标指针递增；当 DF= 1 时，目标指针递减。CLD 指令可以将 DF 清零。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

STOS：存储字符串

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>        IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
STOSB
STOSW
STOSD    386+
```

举例

```
STOSB          ;将 AL 存放到[EDI]中
STOSW          ;将 AX 存放到[EDI]中
STOSD          ;将 EAX 存放到[EDI]中
REP STOSW      ;将 EAX 存放到[EDI]以及它后面的地址，重复 ECX 次
```

说明

STOS 只是将 AL(对于 8 位存储操作而言)，AX(对于 16 位操作而言)，或者 EAX(对于 32 位操作而言)存储到 [DI]或[EDI]所指定的位置。在 16 位模式下，ES 必须包含源操作数的段地址，并且不能被覆盖。在 32 位保护模式下，由于所有的段都是一致的，因此 ES 并不需要被明确地指定。类似的，DI 或 EDI 必须始终是目标操作数的偏移地址。

通过将需要重复的操作次数(不是字节数！)放到 CX/ECX 中，并且在助记符前面加上 rep 前缀，STOS 能够像自动“机枪”一样，自动地将 AL/AX/EAX 存储到以[EDI]为起点的连续内存位置中。每次存储之后，DI/EDI 都被调整(至于如何调整，参见下一自然段)或者 1(如果是 8 位存储操作)，或者 2(如果是 16 位存储操作)，或者 4(如果是 32 位存储操作)，并且 CX/ECX 被递减 1。不要忘记，CX/ECX 是对操作次数(即：数据项存储到内存中的次数)而不是对字节数进行计数！

如果方向标志被(CLD 指令)清零，则调整的意思是递增；如果方向标志已经被(STD 指令)设置，则调整的意思是递减。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS         r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

SUB：算数减法

影响的标志

```
O D I T S Z A P C    OF：溢出标志    TF：陷阱标志    AF：辅助进位标志
F F F F F F F F F    DF：方向标志    SF：符号标志    PF：奇偶标志
*       * * * * *    IF：中断标志    ZF：零标志      CF：进位标志
```

合法形式

```
SUB r8,r8
SUB m8,r8
SUB r8,m8
SUB r16,r16
SUB m16,r16
SUB r16,m16
SUB r32,r32        386+
SUB m32,r32        386+
SUB r32,m32        386+
SUB r8,i8
SUB m8,i8
SUB r16,i16
SUB m16,i16
SUB r32,i32        386+
SUB m32,i32        386+
SUB r16,i8
SUB m16,i8
SUB r32,i8         386+
SUB m32,i8         386+
SUB AL,i8
SUB AX,i16
SUB EAX,i32        386+
```

举例

```
SUB BX,DI
SUB AX,0FFFFH       ;使用单字节操作码
SUB AL,42H          ;使用双字节操作码
SUB BP,17H
SUB ECX,DWORD [ESI+5]
SUB EAX,17
SUB WORD [BX+SI+Inset],5
SUB WORD [ES:BX],0B800H
```

说明

SUB 执行不带借位的减法，将源操作数从目标操作数中减掉，用结果替换目标操作数。如果结果为负，进位标志 CF 被设置。可以通过在 SUB 后面跟上 SBB(带借位的减法)来实现多精度减法，它将进位标志当作为一个算术借位来进行处理。

```
r8 = AL AH BL BH CL CH DL DH   r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS          r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据               m16 = 16 位内存数据
m32 = 32 位内存数据             i8 = 8 位立即数
i16 = 16 位立即数               i32 = 32 位立即数
d8 = 8 位有符号位移             d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

XCHG：交换操作数指令

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>        IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
XCHG r8,r8
XCHG r8,m8
XCHG r16,r16
XCHG r16,m16
XCHG r32,r32   386+
XCHG r32,m32   386+
```

举例

```
XCHG AL,DH
XCHG BH,BYTE [SI]
XCHG SP,BP
XCHG DX,WORD [DI]
XCHG ESI,EDI
XCHG ECX,DWORD [EBP+38]
XCHG AX,BX          ; 使用单字节操作码
```

说明

XCHG 交换它的两个操作数的内容。这就是 XCHG 为什么不具有相同操作数的使用形式的原因，也就是说，“XCHG AX,AX”不是合法形式，因为将一个寄存器与它本身交换根本没有任何逻辑意义。(不过，CPU 执行“XCHG AX,AX”，将其作为无操作指令，或 NOP 的内部等价。)

将一个操作数与 AX 交换只需一个单字节操作码即可实现，所以可以节省取

指时间和代码空间。所有优秀的汇编编译器都识别这些情况，并对它们进行优化，但是如果你正在为一些高级语言手工编译内嵌汇编语句，一定要牢记这种单字节特例。

```
r8 = AL AH BL BH CL CH DL DH r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS       r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据             m16 = 16 位内存数据
m32 = 32 位内存数据           i8 = 8 位立即数
i16 = 16 位立即数             i32 = 32 位立即数
d8 = 8 位有符号位移           d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

XLAT：通过查表来翻译字节

影响的标志

```
O D I T S Z A P C   OF: 溢出标志   TF: 陷阱标志   AF: 辅助进位标志
F F F F F F F F F   DF: 方向标志   SF: 符号标志   PF: 奇偶标志
      <none>        IF: 中断标志   ZF: 零标志     CF: 进位标志
```

合法形式

```
XLAT
XLATB        ; (同义词)
```

举例

```
XLAT
```

说明

XLAT 对 AL 中的 8 位值执行查表翻译。所有的操作数都是隐式的。AL 中的值被视为对于位于 EBX 包含的内存地址处的表中的索引。当 XLAT 被执行时，位于[EBX+AL]处的值将替换原来 AL 中的值。

位于 EBX 寄存器中的 32 位地址处的表不一定非要 256 个字节长，但是，如果 AL 中的值比这个表的长度大，将导致一个未定义的值放入 AL 中，如果该表位于程序的内存空间边缘的话，会增加额外的段错误可能性。

```
r8 = AL AH BL BH CL CH DL DH r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS       r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据             m16 = 16 位内存数据
m32 = 32 位内存数据           i8 = 8 位立即数
```

i16 = 16 位立即数　　i32 = 32 位立即数
d8 = 8 位有符号位移　　d16 = 16 位有符号位移
d32 = 32 位无符号位移

XOR：异或

影响的标志

O	D	I	T	S	Z	A	P	C			
F	F	F	F	F	F	F	F	F	OF：溢出标志	TF：陷阱标志	AF：辅助进位标志
*				*	*	?	*	*	DF：方向标志	SF：符号标志	PF：奇偶标志
									IF：中断标志	ZF：零标志	CF：进位标志

合法形式

```
XOR r8,r8
XOR m8,r8
XOR r8,m8
XOR r16,r16
XOR m16,r16
XOR r16,m16
XOR r32,r32    386+
XOR m32,r32    386+
XOR r32,m32    386+
XOR r8,i8
XOR m8,i8
XOR r16,i16
XOR m16,i16
XOR r32,i32    386+
XOR m32,i32    386+
XOR AL,i8
XOR AX,i16
XOR EAX,i32 386+
```

举例

```
XOR BX,DI
XOR AX,0FFFFH           ;使用单字节操作码
XOR AL,42H              ;使用双字节操作码
XOR EBX,DWORD [EDI]
XOR WORD [ES:BX],0B800H
XOR WORD [BP+SI],DX
```

说明

XOR 对它的两个操作数执行异或逻辑运算。一旦操作完成，结果将替换目标操作数。XOR 是按位执行的，源操作数的第 0 位与目标操作数的第 0 位相异或，源操作数的第 1 位与目标操作数的第 1 位相异或，以此类推。如果操作数不同，异或操作产生一个 1，如果操作数相同，异或操作则产生一个 0。请注意，异或使辅助进位标志 AF 变为不确定状态。CF 和 OF 被清零，其他受影响的标志则根据操作的结果进行设置。

```
r8 = AL AH BL BH CL CH DL DH  r16 = AX BX CX DX BP SP SI DI
sr = CS DS SS ES FS GS        r32 = EAX EBX ECX EDX EBP ESP ESI EDI
m8 = 8 位内存数据             m16 = 16 位内存数据
m32 = 32 位内存数据           i8 = 8 位立即数
i16 = 16 位立即数             i32 = 32 位立即数
d8 = 8 位有符号位移           d16 = 16 位有符号位移
d32 = 32 位无符号位移
```

附录 B

字符集图

本附录包含了 PC 兼容机上常用的两个字符集摘要。第一个为 IBM - 850 字符集，它常用于 Linux 终端实用工具，如 Konsole 和 GNOME 上。第二个是老式的“代码页 437”集合，它基本上是编码到 IBM 兼容 PC 机的 BIOS ROM 中在字符集。

对于每一个集合中的每一个字符，都有一个字形块。每个字形块包括以下信息：

- 字符编号的 3 位十进制形式，从 000 到 255。这些都在每个块的右上角。
- 字符编号的十六进制形式，从 00 到 FF。这些都在每个块的左下角。
- 字符字形是在块的中心。
- 对于从 0-31 的控制字符，该控制字符的名称(例如，NAK、DLE、CR 等)被垂直印刷在块的右下角。

请注意，在 IBM-850 字符集不是被默认加载到普通的 Linux 终端实用工具中，必须在字符集被显示到终端窗口中之前，从选项或设置菜单中进行明确的选择。要想了解更多关于它的内容，请参阅第 6 章中的“Linux 和终端”一节。

ASCII & PC 扩展字符-代码页 437

000 00 NUL	016 10 ► DLE	032 20	048 30 0	064 40 @	080 50 P	096 60 `	112 70 p	128 80 Ç	144 90 É	160 A0 á	176 B0 ░	192 C0 └	208 D0 ╨	224 E0 α	240 F0 ≡
001 01 ☺ SOH	017 11 ◄ DC1	033 21 !	049 31 1	065 41 A	081 51 Q	097 61 a	113 71 q	129 81 ü	145 91 æ	161 A1 í	177 B1 ▒	193 C1 ┴	209 D1 ╤	225 E1 β	241 F1 ±
002 02 ☻ STX	018 12 ↕ DC2	034 22 "	050 32 2	066 42 B	082 52 R	098 62 b	114 72 r	130 82 é	146 92 Æ	162 A2 ó	178 B2 ▓	194 C2 ┬	210 D2 ╥	226 E2 Γ	242 F2 ≥
003 03 ♥ ETX	019 13 ‼ DC3	035 23 #	051 33 3	067 43 C	083 53 S	099 63 c	115 73 s	131 83 â	147 93 ô	163 A3 ú	179 B3 │	195 C3 ├	211 D3 ╙	227 E3 π	243 F3 ≤
004 04 ♦ EOT	020 14 ¶ DC4	036 24 $	052 34 4	068 44 D	084 54 T	100 64 d	116 74 t	132 84 ä	148 94 ö	164 A4 ñ	180 B4 ┤	196 C4 ─	212 D4 ╘	228 E4 Σ	244 F4 ⌠
005 05 ♣ ENQ	021 15 § NAK	037 25 %	053 35 5	069 45 E	085 55 U	101 65 e	117 75 u	133 85 à	149 95 ò	165 A5 Ñ	181 B5 ╡	197 C5 ┼	213 D5 ╒	229 E5 σ	245 F5 ⌡
006 06 ♠ ACK	022 16 ▬ SYN	038 26 &	054 36 6	070 46 F	086 56 V	102 66 f	118 76 v	134 86 å	150 96 û	166 A6 ª	182 B6 ╢	198 C6 ╞	214 D6 ╓	230 E6 µ	246 F6 ÷
007 07 • BEL	023 17 ↨ ETB	039 27 '	055 37 7	071 47 G	087 57 W	103 67 g	119 77 w	135 87 ç	151 97 ù	167 A7 º	183 B7 ╖	199 C7 ╟	215 D7 ╫	231 E7 τ	247 F7 ≈
008 08 ◘ BS	024 18 ↑ CAN	040 28 (	056 38 8	072 48 H	088 58 X	104 68 h	120 78 x	136 88 ê	152 98 ÿ	168 A8 ¿	184 B8 ╕	200 C8 ╚	216 D8 ╪	232 E8 Φ	248 F8 °
009 09 ○ HT	025 19 ↓ EM	041 29)	057 39 9	073 49 I	089 59 Y	105 69 i	121 79 y	137 89 ë	153 99 Ö	169 A9 ⌐	185 B9 ╣	201 C9 ╔	217 D9 ┘	233 E9 Θ	249 F9 ∙
010 0A ◙ LF	026 1A → SUB	042 2A *	058 3A :	074 4A J	090 5A Z	106 6A j	122 7A z	138 8A è	154 9A Ü	170 AA ¬	186 BA ║	202 CA ╩	218 DA ┌	234 EA Ω	250 FA ·
011 0B ♂ VT	027 1B ← ESC	043 2B +	059 3B ;	075 4B K	091 5B [	107 6B k	123 7B {	139 8B ï	155 9B ¢	171 AB ½	187 BB ╗	203 CB ╦	219 DB █	235 EB δ	251 FB √
012 0C ♀ FF	028 1C ∟ FS	044 2C ,	060 3C <	076 4C L	092 5C \	108 6C l	124 7C |	140 8C î	156 9C £	172 AC ¼	188 BC ╝	204 CC ╠	220 DC ▄	236 EC ∞	252 FC ^{n}
013 0D ♪ CR	029 1D ↔ GS	045 2D -	061 3D =	077 4D M	093 5D]	109 6D m	125 7D }	141 8D ì	157 9D ¥	173 AD ¡	189 BD ╜	205 CD ═	221 DD ▌	237 ED Ø	253 FD ^{2}
014 0E ♫ SO	030 1E ▲ RS	046 2E .	062 3E >	078 4E N	094 5E ^	110 6E n	126 7E ~	142 8E Ä	158 9E ₧	174 AE «	190 BE ╛	206 CE ╬	222 DE ▐	238 EE ε	254 FE ■
015 0F ☼ SI	031 1F ▼ US	047 2F /	063 3F ?	079 4F O	095 5F _	111 6F o	127 7F ⌂	143 8F Å	159 9F ƒ	175 AF »	191 BF ┐	207 CF ╧	223 DF ▀	239 EF ∩	255 FF

ASCII & PC 扩展字符 - IBM-850

000 NUL 00	016 ► DLE 10	032 20	048 0 30	064 @ 40	080 P 50	096 ` 60	112 p 70	128 Ç 80	144 É 90	160 á A0	176 ░ B0	192 └ C0	208 ð D0	224 Ó E0	240 − F0
001 ☺ SOH 01	017 ◄ DC1 11	033 ! 21	049 1 31	065 A 41	081 Q 51	097 a 61	113 q 71	129 ü 81	145 æ 91	161 í A1	177 ▒ B1	193 ┴ C1	209 Ð D1	225 ß E1	241 ± F1
002 ● STX 02	018 ↕ DC2 12	034 " 22	050 2 32	066 B 42	082 R 52	098 b 62	114 r 72	130 é 82	146 Æ 92	162 ó A2	178 ▓ B2	194 ┬ C2	210 Ê D2	226 Ô E2	242 = F2
003 ♥ ETX 03	019 ‼ DC3 13	035 # 23	051 3 33	067 C 43	083 S 53	099 c 63	115 s 73	131 â 83	147 ô 93	163 ú A3	179 │ B3	195 ├ C3	211 Ë D3	227 Ò E3	243 ¾ F3
004 ♦ EOT 04	020 ¶ DC4 14	036 $ 24	052 4 34	068 D 44	084 T 54	100 d 64	116 t 74	132 ä 84	148 ö 94	164 ñ A4	180 ┤ B4	196 ─ C4	212 ı D4	228 õ E4	244 ¶ F4
005 ♣ ENQ 05	021 § NAK 15	037 % 25	053 5 35	069 E 45	085 U 55	101 e 65	117 u 75	133 à 85	149 ò 95	165 Ñ A5	181 Á B5	197 ┼ C5	213 í D5	229 Õ E5	245 § F5
006 ♠ ACK 06	022 ▬ SYN 16	038 & 26	054 6 36	070 F 46	086 V 56	102 f 66	118 v 76	134 å 86	150 û 96	166 ª A6	182 Â B6	198 ã C6	214 Î D6	230 µ E6	246 ÷ F6
007 • BEL 07	023 ↨ ETB 17	039 ' 27	055 7 37	071 G 47	087 W 57	103 g 67	119 w 77	135 ç 87	151 ù 97	167 º A7	183 À B7	199 Ã C7	215 Ï D7	231 þ E7	247 ¸ F7
008 ◘ BS 08	024 ↑ CAN 18	040 (28	056 8 38	072 H 48	088 X 58	104 h 68	120 x 78	136 ê 88	152 ÿ 98	168 ¿ A8	184 © B8	200 ╚ C8	216 ┼ D8	232 Þ E8	248 ° F8
009 ○ HT 09	025 ↓ EM 19	041) 29	057 9 39	073 I 49	089 Y 59	105 i 69	121 y 79	137 ë 89	153 Ö 99	169 ® A9	185 ╣ B9	201 ╔ C9	217 ┘ D9	233 Ú E9	249 ¨ F9
010 ■ LF 0A	026 → SUB 1A	042 * 2A	058 : 3A	074 J 4A	090 Z 5A	106 j 6A	122 z 7A	138 è 8A	154 Ü 9A	170 ¬ AA	186 ║ BA	202 ╩ CA	218 ┌ DA	234 Û EA	250 · FA
011 ♂ VT 0B	027 ← ESC 1B	043 + 2B	059 ; 3B	075 K 4B	091 [5B	107 k 6B	123 { 7B	139 ï 8B	155 ø 9B	171 ½ AB	187 ╗ BB	203 ╦ CB	219 █ DB	235 Ù EB	251 ¹ FB
012 ♀ FF 0C	028 ∟ FS 1C	044 , 2C	060 < 3C	076 L 4C	092 \ 5C	108 l 6C	124 | 7C	140 î 8C	156 £ 9C	172 ¼ AC	188 ╝ BC	204 ╠ CC	220 ▄ DC	236 ý EC	252 ³ FC
013 ♪ CR 0D	029 ↔ GS 1D	045 - 2D	061 = 3D	077 M 4D	093] 5D	109 m 6D	125 } 7D	141 ì 8D	157 Ø 9D	173 ¡ AD	189 ¢ BD	205 ═ CD	221 ¦ DD	237 Ý ED	253 ² FD
014 ♫ SO 0E	030 ▲ RS 1E	046 . 2E	062 > 3E	078 N 4E	094 ^ 5E	110 n 6E	126 ~ 7E	142 Ä 8E	158 × 9E	174 « AE	190 ¥ BE	206 ╬ CE	222 Ì DE	238 ¯ EE	254 ■ FE
015 ☼ SI 0F	031 ▼ US 1F	047 / 2F	063 ? 3F	079 O 4F	095 _ 5F	111 o 6F	127 ⌂ 7F	143 Å 8F	159 ƒ 9F	175 » AF	191 ┐ BF	207 ¤ CF	223 ▀ DF	239 ´ EF	255 FF

Assembly Language
Step-By-Step: Programming with Linux 3rd Edition

汇编语言：基于Linux环境

学好汇编语言，才算真正了解计算机

作为CPU的“母语”，汇编语言是其他所有编程语言（尤其是C，C++和Pascal）的基础。可以这么说，掌握了汇编语言，程序员就能“打通任督二脉”，轻轻松松掌握其他语言。然而，汇编语言的艰深晦涩往往让人望而却步，被误认为是少数极客独享的“专利”。

为破除汇编语言的神秘感，畅销书作者Jeff Duntemann特别针对没有任何编程经验的读者，沿袭其诙谐生动的行文风格，抽丝剥茧，条分缕析，在全新修订的《汇编语言：基于Linux环境》中透彻讲解Intel x86计算机的工作机理。

本书主题

- 回顾计算和编程背后的基础概念，包括十六进制和二进制
- 了解Intel CPU的演变和现代x86处理器的工作原理
- 掌握编程过程（编辑源代码、汇编、连接和调试等）
- 解读x86 32位保护模式内存寻址
- 通过剖析大量完整的范例来学习x86指令集
- 学习使用Ubuntu Linux下丰富的免费编程实用工具，包括Kate编辑器、NASM 汇编器和 GNU 工具集
- 掌握Linux编程的实用细节，包括进程、宏、INT 80h调用门和标准C库调用

作者介绍

Jeff Duntemann
从事计算机相关文章和书籍的写作三十余年，主题涉及编程、无线网络和系统管理等。他曾担任过知名IT杂志*Dr. Dobb's*的专栏作家，历任很多知名电脑编程杂志的编辑。在工作之余，他爱好天文和无线电，也喜欢写博客和科幻小说。

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

WILEY

ISBN 978-7-302-34592-3
定价：99.00元